



Using OpenFlow 1.3

RYU SDN Framework

RYU project team

Contents

1	서문	1
2	스위칭 허브	3
2.1	스위칭 허브	3
2.2	OpenFlow 의 한 스위칭 허브	3
2.3	Ryu를 사용한 스위칭 허브 구현	5
2.4	Ryu 응용 프로그램 실행	13
2.5	정리	18
3	트래픽 모니터	19
3.1	네트워크 정기 검사	19
3.2	트래픽 모니터 구현	19
3.3	트래픽 모니터 실행	25
3.4	정리	26
4	REST연계	27
4.1	REST API의 기본	27
4.2	REST API된 스위칭 허브 구현	27
4.3	SimpleSwitchRest13 클래스의 구현	29
4.4	SimpleSwitchController 클래스의 구현	30
4.5	REST API 탑재 스위칭 허브의 실행	32
4.6	정리	33
5	링크 어그리게이션	35
5.1	링크 어그리게이션	35
5.2	Ryu 응용 프로그램의 실행	35
5.3	Ryu의 링크 애그리 게이션 기능의 구현	46
5.4	정리	54
6	스패닝 트리	55
6.1	스패닝 트리	55
6.2	Ryu 응용 프로그램의 실행	57
6.3	OpenFlow 스패닝 트리	67
6.4	Ryu 스패닝 트리 구현	68
6.5	정리	76
7	IGMP 스누핑	79
7.1	IGMP 스누핑	79

7.2 Ryu 응용 프로그램의 실행	83
7.3 Ryu에 따르면 IGMP 스누핑 기능의 구현	95
8 OpenFlow 프로토콜	105
8.1 패치	105
8.2 지침	106
8.3 액션	106
9 ofproto 라이브러리	109
9.1 개요	109
9.2 모듈 구성	109
9.3 기본적인 사용법	110
10 패킷 라이브러리	113
10.1 기본적인 사용법	113
10.2 어플리케이션 예	115
11 OF-Config 라이브러리	119
11.1 OF-Config 프로토콜	119
11.2 라이브러리 구성	119
11.3 예제	120
12 방화벽	121
12.1 단일 거주자의 동작 예	121
12.2 멀티 테넌트의 동작 예	129
12.3 REST API 목록	133
13 라우터	137
13.1 단일 거주자의 동작 예	137
13.2 멀티 테넌트의 동작 예	146
13.3 REST API 목록	158
14 OpenFlow 스위치 테스트 도구	161
14.1 테스트 도구의 개요	161
14.2 사용 방법	162
14.3 테스트 도구 사용 예	165
15 아키텍처	173
15.1 응용 프로그래밍 모델	173
16 컨트리뷰션	175
16.1 개발 체제	175
16.2 개발 환경	175
16.3 패치 쓰기	176
17 도입 사례	177
17.1 Stratosphere SDN Platform (스트라토스 피어)	177
17.2 SmartSDN Controller (NTT 콤웨어)	177

서문

이 책은 Software Defined Networking (SDN)을 실현하기 위한 개발 프레임워크인 Ryu에 관한 전문 서적입니다.

왜 Ryu일까요?

이 문서에서 답변을 찾을 수 있기를 바랍니다.

제 1 장 ~ 제 5 장 순서로 읽어가는 것이 좋습니다. 제 1 장에서는 간단한 스위치 허브를 구현하고, 다음 장에서 트래픽 모니터링 및 링크 어그리게이션 등의 기능을 추가로 구현합니다. 실제 예제를 통해 Ryu를 사용한 프로그래밍을 소개합니다.

제 6 장 ~ 제 9 장에서는 Ryu를 사용한 프로그래밍에 필요한, OpenFlow 프로토콜이나 패킷 라이브러리들을 자세히 소개합니다. 그 다음 10 장 ~ 12 장에서는 Ryu 샘플 응용 프로그램으로 포함되어 있는 방화벽 및 테스트 도구의 사용 방법을 소개합니다. 마지막으로 제 13 장 ~ 제 15 장에서는 Ryu의 아키텍처 및 도입 사례에 대해 소개 합니다.

마지막으로, Ryu 프로젝트를 지원 해주신 분들에게 감사하고 싶습니다. 특히, 사용자 여러분 감사합니다. 메일링리스트에서 여러분의 의견을 대기하고 있습니다.

함께 Ryu를 개발합시다!

스위칭 허브

이 장에서는 간단한 스위칭 허브 구현을 소재로 Ryu를 사용한 응용 프로그램을 구현하는 방법을 설명하고 있습니다.

2.1 스위칭 허브

스위칭 허브는 다양한 기능들을 갖고 있습니다만, 여기에서는 다음과 같은 간단한 기능을 가진 스위칭 허브의 구현을 살펴 보고자 합니다.

- 포트에 연결되어 있는 호스트의 MAC 주소를 학습하고 MAC 주소 테이블을 유지하기
- 이미 학습된 호스트에 대한 패킷을 수신하면 호스트에 연결된 포트로 전송
- 알 수 없는 호스트에 대한 패킷을 수신하면, Flooding

이러한 스위치를 Ryu를 사용하여 구현해 봅시다.

2.2 OpenFlow 의한 스위칭 허브

OpenFlow 스위치는 Ryu와 같은 OpenFlow 컨트롤러의 지시를 받고, 다음과 같은 것들을 수행할 수 있습니다.

- 수신된 패킷의 주소를 재작성(rewrite)하거나 지정된 포트쪽으로부터 전송
- 받은 패킷을 컨트롤러에 전송 (Packet-In)
- 컨트롤러에 의해 전달된 (forwarded) 패킷을 특정 포트쪽으로부터 전송 (Packet-Out)

이러한 기능들을 결합하여 스위칭 허브를 만들 수 있습니다.

우선, Packet-In 기능을 이용하여 MAC 주소 학습할 필요가 있습니다. 컨트롤러는 Packet-In 기능을 이용하여 스위치로부터 패킷을 받을 수가 있습니다. 스위치는 받은 패킷을 분석하고 연결되어 있는 포트에 대한 호스트의 MAC 주소 및 정보를 학습 할 수 있습니다.

학습 후에는 받은 패킷을 전송합니다. 스위치는 패킷의 목적지 MAC 주소가 학습된 호스트에 속해 있는지 아닌지를 찾아봅니다. 검색 결과 여부에 따라 스위치는 다음과 같은 작업을 수행합니다.

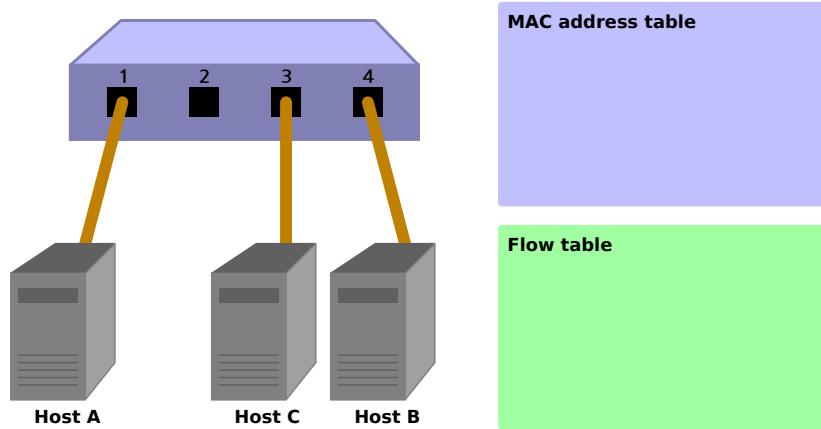
- 학습된 호스트인 경우 ... Packet-Out 기능으로 연결된 포트쪽으로 패킷을 전송
- 알 수 없는 호스트인 경우 ... Packet-Out 기능으로 패킷을 플러딩 (Flooding)

이러한 동작을 그림과 함께 단계별로 설명합니다.

1. 초기 상태

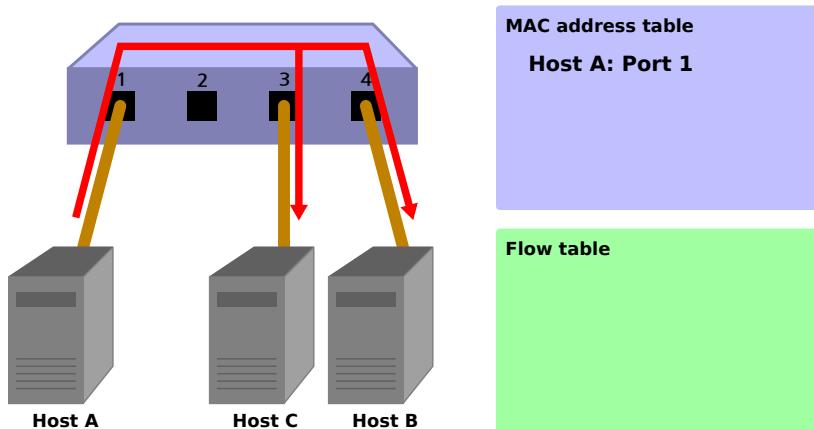
플로우 테이블이 비어 있는 초기 상태입니다.

포트 1에 호스트 A, 포트 4에 호스트 B, 포트 3에 호스트 C가 연결되어 있다고 가정합니다.



2. 호스트 A → 호스트 B

호스트 A에서 호스트 B로 패킷이 전송되면 Packet-In 메시지가 전송되고 호스트 A의 MAC 주소가 포트 1에 학습됩니다. 호스트 B의 포트는 아직 알지 못하기 때문에 패킷은 플러딩되고 따라서 해당 패킷은 호스트 B와 호스트 C에서 수신됩니다.



Packet-In:

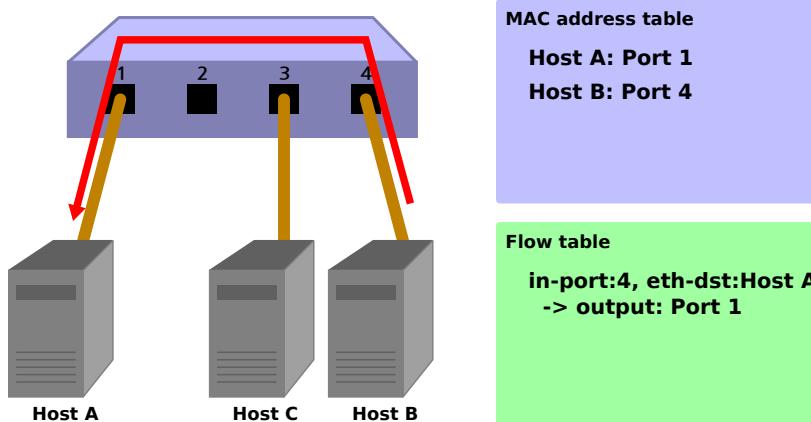
```
in-port: 1
eth-dst: 호스트 B
eth-src: 호스트 A
```

Packet-Out:

```
action: OUTPUT:Flooding
```

3. 호스트 B→호스트 A

호스트 B에서 호스트 A로 패킷이 리턴되면 플로우 테이블에 항목을 추가하고 또한 패킷은 포트 1에 전송됩니다. 따라서 호스트 C는 이 패킷을 수신하지 않습니다.



Packet-In:

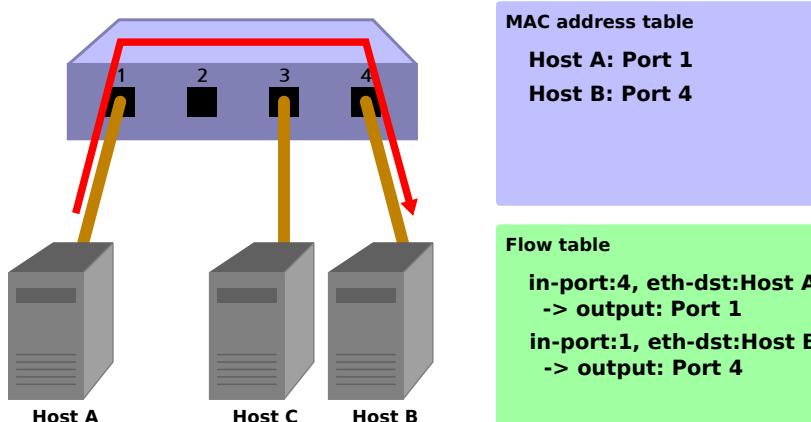
```
in-port: 4
eth-dst: 호스트A
eth-src: 호스트B
```

Packet-Out:

```
action: OUTPUT:포트 1
```

4. 호스트A→호스트B

또한, 호스트 A에서 호스트 B로 패킷이 전송되면 플로우 테이블에 항목을 추가하고 또한 패킷은 포트 4에 전송됩니다.



Packet-In:

```
in-port: 1
eth-dst: 호스트B
eth-src: 호스트A
```

Packet-Out:

```
action: OUTPUT:포트 4
```

이제, Ryu를 사용하여 구현된 스위칭 허브 소스 코드를 살펴 보겠습니다.

2.3 Ryu를 사용한 스위칭 허브 구현

스위칭 허브에 대한 소스 코드는 Ryu 소스 트리에 있습니다.

[ryu/app/simple_switch_13.py](#)

OpenFlow 버전에 따라 그 밖에도 simple_switch.py (OpenFlow 1.0), simple_switch_12.py (OpenFlow 1.2)이 있지만, 여기에서는 OpenFlow 1.3을 지원하는 구현을 살펴 보겠습니다.

짧은 소스 코드이므로, 전체를 여기에 게재합니다.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPIInstructionActions(ofproto.OFPI_APPLY_ACTIONS,
                                              actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                               match=match, instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ether.ethernet)[0]

        dst = eth.dst
        src = eth.src
```

```

dpid = datapath.id
self.mac_to_port.setdefault(dpid, {})

self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port

if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]

# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
    self.add_flow(datapath, 1, match, actions)

data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                         in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)

```

그러면, 각각의 구현 내용을 살펴 보겠습니다.

2.3.1 클래스의 정의 및 초기화

Ryu 응용 프로그램으로 구현하기 위해 ryu.base.app_manager.RyuApp을 상속합니다. 또한 OpenFlow 1.3을 사용하기 때문에 OFP_VERSIONS에 OpenFlow 1.3 버전을 지정합니다.

또한 MAC 주소 테이블에 해당하는 mac_to_port를 정의합니다.

OpenFlow 프로토콜은 OpenFlow 스위치와 컨트롤러가 통신을 위해 필요한 핸드 셰이크 등의 몇 가지 단계가 정의되어 있습니다. 그러나, Ryu의 프레임워크가 이러한 단계들을 다루기에, Ryu 응용 프로그램에서 이러한 것들을 신경쓸 필요가 없습니다.

```

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    ...

```

2.3.2 이벤트 처리기

Ryu에서, OpenFlow 메시지를 수신하면 해당 메시지에 대응하는 이벤트가 생성됩니다. Ryu 응용 프로그램은 수신하고자 하는 메시지에 대응하는 이벤트 처리기를 구현합니다.

이벤트 처리기는 인수 처리를 위해 이벤트 객체를 갖는 함수를 정의하고 한정(decorate)을 위해 ryu.controller.handler.set_ev_cls 한정자를 사용합니다.

set_ev_cls는 수신하는 메시지를 지원하는 이벤트 클래스와 인수에 대한 OpenFlow 스위치의 상태를 지정합니다.

이벤트 클래스 이름은 `ryu.controller.ofp_event.EventOFPPacketIn`입니다. 예를 들어, Packet-In 메시지의 경우 `EventOFPPacketIn`입니다. 자세한 내용은 Ryu 문서 API 레퍼런스를 참조하십시오. 상태에 대해서는 다음 중 하나 또는 리스트로 지정합니다.

정의	설명
<code>ryu.controller.handler.HANDSHAKE_DISPATCHER</code>	HELLO 메시지 교환
<code>ryu.controller.handler.CONFIG_DISPATCHER</code>	SwitchFeatures 메시지의 수신
<code>ryu.controller.handler.MAIN_DISPATCHER</code>	표준 상태
<code>ryu.controller.handler.DEAD_DISPATCHER</code>	연결 절단

Table-miss 흐름 항목 추가

OpenFlow 스위치와의 핸드 셰이크가 완료된 후, Table-miss 플로우 항목(entry)이 플로우 테이블에 추가되고 Packet-In 메시지를 수신할 준비를 합니다.

구체적으로는 Switch Features (Features Reply) 메시지를 수신하자마자 Table-miss 플로우 항목을 추가합니다.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # ...

ev.msg에는 이벤트에 해당하는 OpenFlow 메시지 클래스의 인스턴스가 저장되어 있습니다. 이 경우에는 ryu.ofproto.ofproto_v1_3_parser.OFPSwitchFeatures입니다.
```

`msg.datapath`에는 이 메시지를 발행한 OpenFlow 스위치에 해당하는 `ryu.controller.controller.Datapath` 클래스의 인스턴스가 저장되어 있습니다.

`Datapath` 클래스는 OpenFlow 스위치와의 실제 통신 처리 및 수신 메시지에 대응하는 이벤트 발행 등의 중요한 작업을 수행하고 있습니다.

Ryu 응용 프로그램에서 사용되는 주요 특성은 다음과 같습니다.

속성 이름	설명
<code>id</code>	연결된 OpenFlow 스위치 ID (데이터 경로 ID)입니다.
<code>ofproto</code>	사용하는 OpenFlow 버전에 대응하는 <code>ofproto</code> 모듈을 보여줍니다. 현재는 다음 중 하나입니다. <code>ryu.ofproto.ofproto_v1_0</code> <code>ryu.ofproto.ofproto_v1_2</code> <code>ryu.ofproto.ofproto_v1_3</code> <code>ryu.ofproto.ofproto_v1_4</code>
<code>ofproto_parser</code>	<code>ofproto</code> 와 마찬가지로 <code>ofproto_parser</code> 모듈을 보여줍니다. 현재는 다음 중 하나입니다. <code>ryu.ofproto.ofproto_v1_0_parser</code> <code>ryu.ofproto.ofproto_v1_2_parser</code> <code>ryu.ofproto.ofproto_v1_3_parser</code> <code>ryu.ofproto.ofproto_v1_4_parser</code>

Ryu 응용 프로그램에서 사용하는 `Datapath` 클래스의 주요 메서드는 다음과 같습니다.

`send_msg(msg)`

OpenFlow 메시지를 보냅니다. `msg`는 보내는 OpenFlow 메시지에 대응하는 `ryu.ofproto.ofproto_parser.MsgBase`의 서브 클래스입니다.

스위칭 허브는 받은 Switch Features 메시지 자체는 특별히 사용하지 않습니다. Table-miss 플로우 항목을 추가하는 타이밍을 위한 이벤트로 다루고 있습니다.

```
def switch_features_handler(self, ev):
    # ...
```

```

# install table-miss flow entry
#
# We specify NO_BUFFER to max_len of the output action due to
# OVS bug. At this moment, if we specify a lesser number, e.g.,
# 128, OVS will send Packet-In with invalid buffer_id and
# truncated packet data. In that case, we cannot output packets
# correctly.
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                  ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)

```

Table-miss 플로우 항목은 우선 순위가 최저(0)이고, 모든 패킷에 매치되는 항목입니다. 이 항목의 명령(instruction)에는 컨트롤러 포트로의 출력을 출력 액션으로 지정하여, 들어오는 패킷이 모든 정상(normal) 플로우 항목과 일치하지 않으면, Packet-In을 생성할 수 있습니다.

주석: 2014년 1월 현재 Open vSwitch는 OpenFlow 1.3의 지원이 불완전이며, OpenFlow 1.3 이전과 마찬가지로 기본적으로 Packet-In이 생성됩니다. 또한 Table-miss 플로우 항목이 현재는 지원되지 않고 정상(normal) 플로우 항목으로 취급됩니다.

모든 패킷에 매치시키기 위해 빈 Match를 생성합니다. Match는 OFPMatch 클래스로 표현됩니다.

그 다음, 컨트롤러 포트로 전송하는 OUTPUT 액션 클래스(OFPActionOutput)의 인스턴스를 생성합니다. 컨트롤러가 output 대상으로 지정되고 max_len에는 OFPCML_NO_BUFFER을 지정하여 모든 패킷들이 컨트롤러로 전송되도록 합니다.

주석: 컨트롤러는 패킷의 시작 부분(Ethernet 헤더 분)만을 전송하고 나머지는 스위치 버퍼에 두는 것이 효율성 측면에서 바람직 하지만 Open vSwitch 버그(2014년 1월 현재)를 해결하기 위해 여기에 전체 패킷을 전송합니다.

마지막으로, 우선 순위 0(가장 낮음)을 지정하여 add_flow() 메소드를 실행하여 Flow Mod 메시지를 보냅니다. add_flow() 메서드의 내용에 대해서는 뒤에서 설명하고자 합니다.

Packet-in 메시지

알 수 없는 목적지를 가진 수신 패킷을 허용하기 위해 Packet-In 이벤트 처리기를 만듭니다.

```

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    #

```

자주 사용되는 OFPPacketIn 클래스의 속성은 다음과 같은 것들이 있습니다.

속성 이름	설명
match	ryu.ofproto.ofproto_v1_3_parser.OFPMatch 클래스의 인스턴스에서 들어오는 패킷의 메타 정보가 설정되어 있습니다.
data	수신 패킷 자체를 나타내는 이진 데이터입니다.
total_len	수신 패킷의 데이터 길이입니다.
buffer_id	수신 패킷이 OpenFlow 스위치에서 버퍼처리 되는 경우 해당 ID가 표시됩니다. 버퍼처리 되지 않는 경우 ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER 가 설정됩니다.

MAC 주소 테이블 업데이트

```

def _packet_in_handler(self, ev):
    #
    in_port = msg.match['in_port']

```

```

pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]

dst = eth.dst
src = eth.src

dpid = datapath.id
self.mac_to_port.setdefault(dpid, {})

self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port

# ...

```

OFPPacketIn 클래스의 match에서 수신 포트(in_port)를 가져옵니다. 대상 MAC 주소와 원본 MAC 주소는 Ryu 패킷 라이브러리를 사용하여 수신 패킷의 Ethernet 헤더에서 얻어집니다.

가져온 원본 MAC 주소와 수신 포트 번호를 기반으로 MAC 주소 테이블을 업데이트합니다.

여러 OpenFlow 스위치와의 연결에 대응하기 위해 MAC 주소 테이블은 OpenFlow 스위치마다 관리하도록 되어 있습니다. OpenFlow 스위치를 식별하는 데이터 경로 ID를 이용하고 있습니다.

대상 포트 판정

대상 MAC 주소가 MAC 주소 테이블에 존재하는 경우 대응되는 포트 번호가 사용됩니다. 발견되지 않으면 플러딩(OFPP_FLOOD)를 출력 포트에 지정하는 OUTPUT 액션 클래스의 인스턴스를 생성합니다.

```

def _packet_in_handler(self, ev):
    # ...

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    # ...

```

대상 MAC 주소가 있으면, OpenFlow 스위치의 플로우 테이블에 항목을 추가합니다.

Table-miss 플로우 항목의 추가와 마찬가지로 매치와 액션을 지정하고 add_flow()를 실행하여 플로우 항목을 추가합니다.

Table-miss 플로우 항목과 달리, 이번에는 매치 조건을 설정합니다. 이번 스위칭 허브의 구현에서는 수신 포트 (in_port)와 대상 MAC 주소 (eth_dst)를 지정합니다. 예를 들어, 「포트 1에서 수신하고 호스트 B로 향하는」 패킷이 대상이 됩니다.

이번 플로우 항목은 우선 순위에 1을 지정합니다. 값이 클 수록 우선 순위가 높아지므로 여기에 추가하는 플로우 항목은 Table-miss 플로우 항목보다 먼저 평가됩니다.

위의 작업을 포함하여 정리하면 다음과 유사한 항목을 플로우 테이블에 추가합니다.

포트 1에서 수신한 호스트 B로 전달되는 (대상 MAC 주소가 B) 패킷을 포트 4에 전송하기

힌트: OpenFlow는 NORMAL 포트는 논리적인 출력 포트가 옵션으로 규정 되고 출력 포트에 NORMAL을 지정하면 스위치의 L2/L3 기능을 사용 하라고 패킷을 처리할 수 있습니다. 즉, 모든 패킷을 NORMAL 포트에 출력하도록 지시하는 것만으로, 스위칭 허브 역할을 하는 것처럼 할 수 있지만, 여기에서는 각각의 처리를 OpenFlow를 사용하여 수행하는 것으로 합니다.

플로우 항목의 추가 처리

Packet-In 처리기에서의 처리가 아직 끝나지 않지만 여기서 일단 플로우 항목을 추가하는 메서드 쪽을 살펴 보겠습니다.

```
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    # ...
```

플로우 항목에는 대상 패킷의 조건을 나타내는 매치와 패킷에 대한 작업을 나타내는 인스트럭션, 우선 순위, 유효 시간 등을 설정합니다.

스위칭 허브의 구현은 인스트럭션에 Apply Actions를 사용하여 지정된 액션을 즉시 적용하도록 설정합니다.

마지막으로, Flow Mod 메시지를 발행하여 플로우 테이블에 항목을 추가합니다.

```
def add_flow(self, datapath, port, dst, actions):
    # ...

    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)
    datapath.send_msg(mod)
```

Flow Mod 메시지에 대응하는 클래스는 OFPFlowMod 클래스입니다. OFPFlowMod 클래스의 인스턴스를 생성하여 Datapath.send_msg() 메서드를 사용해 OpenFlow 스위치에 메시지를 보냅니다.

OFPFlowMod 클래스의 생성자에는 많은 인수가 있습니다만, 일반적으로 대부분의 경우 기본값을 그대로 하면 됩니다. 괄호 안은 기본값입니다.

datapath

플로우 테이블을 조작하는 대상 OpenFlow 스위치에 해당하는 Datapath 클래스의 인스턴스입니다. 일반적으로 Packet-In 메시지 등의 처리기 에 전달되는 이벤트에서 가져온 것입니다.

cookie (0)

컨트롤러에 지정하는 선택적 값으로 항목을 업데이트 또는 삭제할 때 필터 조건으로 사용할 수 있습니다. 패킷 처리에는 사용되지 않습니다.

cookie_mask (0)

항목의 업데이트 또는 삭제하는 경우 0이 아닌 값을 지정하면 항목의 cookie 값을 사용하는 동작 대상 항목의 필터로 사용됩니다.

table_id (0)

동작 대상의 플로우 테이블의 테이블 ID를 지정합니다.

command (ofproto_v1_3.OFPFC_ADD)

어떤 작업을 할 것인지를 지정합니다.

값	설명
OFPFC_ADD	새로운 플로우 항목을 추가합니다
OFPFC MODIFY	플로우 항목을 업데이트합니다
OFPFC MODIFY_STRICT	엄격하게 일치하는 플로우 항목을 업데이트합니다
OFPFC_DELETE	플로우 항목을 삭제합니다
OFPFC_DELETE_STRICT	엄격하게 일치하는 플로우 항목을 삭제합니다

idle_timeout (0)

해당 항목의 유효 기간을 초 단위로 지정합니다. 항목이 참조되지 않고 idle_timeout에서 지정된 시간을 초과하면 항목이 제거됩니다. 항목이 참조 될 때 경과 시간은 리셋됩니다.

항목이 삭제되면 Flow Removed 메시지가 컨트롤러에 알려 있습니다.

hard_timeout (0)

해당 항목의 유효 기간을 초 단위로 지정합니다. idle_timeout과 달리, hard_timeout은 항목이 참조되더라도 경과 시간은 리셋되지 않습니다. 즉, 항목의 참조 여부에 관계없이 지정된 시간이 경과하면 항목이 삭제됩니다.

idle_timeout과 마찬가지로 항목이 삭제되면 Flow Removed 메시지가 보내집니다.

priority (0)

해당 항목의 우선 순위를 지정합니다. 값이 클수록 우선 순위가 높습니다.

buffer_id (ofproto_v1_3.OFP_NO_BUFFER)

OpenFlow 스위치에서 버퍼된 패킷의 버퍼 ID를 지정합니다. 버퍼 ID는 Packet-In 메시지로 통지되고, 지정하면 OFPP_TABLE을 출력 포트에 지정된 Packet-Out 메시지와 Flow Mod 메시지 두 메시지를 보낸 것처럼 처리됩니다. command가 OFPFC_DELETE 또는 OFPFC_DELETE_STRICT의 경우는 무시됩니다.

버퍼 ID를 지정하지 않으면, OFP_NO_BUFFER 을 설정합니다.

out_port (0)

OFPFC_DELETE 또는 OFPFC_DELETE_STRICT의 경우 대상 항목을 출력 포트 필터링합니다. OFPFC_ADD, OFPFC MODIFY, OFPFC MODIFY_STRICT 의 경우는 무시됩니다.

출력 포트의 필터를 해제하려면 OFPP_ANY 을 지정합니다.

out_group (0)

out_port와 마찬가지로 출력 그룹에서 필터링합니다.

해제하려면 OFPG_ANY 을 지정합니다.

flags (0)

다음 플래그의 조합을 지정할 수 있습니다.

값	설명
OFPFF_SEND_FLOW_REM	FLOW_REM이 항목이 삭제될 때 컨트롤러에 Flow Removed 메시지를 발행합니다.
OFPFF_CHECK_OVERLAP	OFPFC_ADD의 경우 중복 항목의 검사를 수행 합니다. 중복된 항목이 있는 경우에는 Flow Mod가 손실 되고 오류가 반환됩니다.
OFPFF_RESET_COUNTS	해당 항목의 패킷과 바이트 카운터를 재설정합니다.
OFPFF_NO_PKT_COUNTS	이 항목의 패킷 카운터를 해제합니다.
OFPFF_NO_BYT_COUNTS	이 항목에 대한 바이트 카운터를 해제합니다.

match (None)

Match를 지정합니다.

instructions ([])

명령어의 목록을 지정합니다.

패킷 전송

Packet-In 처리기로 돌아가 마지막 처리 단계를 설명합니다.

대상 MAC 주소를 MAC 주소 테이블에서 발견하는 여부와 관계없이 최종 적으로 Packet-Out 메시지를 생성하여 수신 패킷을 전송합니다.

```
def _packet_in_handler(self, ev):
    # ...

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                             in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

Packet-Out 메시지에 대응하는 클래스는 OFPPacketOut 클래스입니다.

OFPPacketOut 생성자의 인수는 다음과 같이되어 있습니다.

datapath

OpenFlow 스위치에 해당하는 Datapath 클래스의 인스턴스를 지정합니다.

buffer_id

OpenFlow 스위치에서 베퍼 된 패킷 베퍼 ID를 지정합니다. 베퍼를 사용하지 않으면, OFP_NO_BUFFER 을 지정합니다.

in_port

패킷을 수신 한 포트를 지정합니다. 수신 패킷이 아닌 경우 OFPP_CONTROLLER 를 지정합니다.

actions

작업 목록을 지정합니다.

data

패킷의 이진 데이터를 지정합니다. buffer_id에 OFP_NO_BUFFER 가 지정된 경우에 사용됩니다. OpenFlow 스위치 베퍼를 사용하는 경우 생략합니다.

스위칭 허브의 구현은 buffer_id에 Packet-In 메시지 buffer_id를 지정합니다. Packet-In 메시지 내 buffer_id 가 무효 인 경우, 들어오는 Packet-In 패킷을 data로 지정하여 패킷을 전송합니다.

이제 스위칭 허브 소스 코드의 설명이 끝났습니다. 다음으로 스위칭 허브를 실행하여 실제 동작을 확인합니다.

2.4 Ryu 응용 프로그램 실행

스위칭 허브의 실행을 위해 OpenFlow 스위치는 Open vSwitch 실행 환경으로 mininet을 사용합니다.

Ryu의 OpenFlow Tutorial VM 이미지가 포함되어 있으므로, 이 VM 이미지를 이용하면 실험 환경을 쉽게 준비할 수 있습니다.

VM 이미지

<http://sourceforge.net/projects/ryu/files/vmimages/OpenFlowTutorial/>

OpenFlow_Tutorial_Ryu3.2.ova (약1.4GB)

관련 문서 (Wiki 페이지)

https://github.com/osrg/ryu/wiki/OpenFlow_Tutorial

문서에 있는 VM 이미지는 Open vSwitch와 Ryu의 버전이 오래 되었기 때문에 주의하시기 바랍니다.
이 VM 이미지를 사용하지 않고, 스스로 환경을 구축하는 것 또한 당연히 가능합니다. 스스로 환경을 구축하고자 하는 경우, 참고로, VM 이미지에서 사용하는 각 소프트웨어 버전은 다음과 같습니다.

Mininet VM 버전 2.0.0 <http://mininet.org/download/>

Open vSwitch 버전 1.11.0 <http://openvswitch.org/download/>

Ryu 버전 3.2 <https://github.com/osrg/ryu/>

```
$ sudo pip install ryu
```

여기에서는 Ryu 용 OpenFlow Tutorial의 VM 이미지를 사용합니다.

2.4.1 Mininet 실행

mininet에서 xterm을 시작하기 위해 X를 사용할 수 있는 환경이 필요합니다.

여기에서는 OpenFlow Tutorial VM을 사용하므로, ssh에서 X11 Forwarding을 사용하여 로그인하십시오.

```
$ ssh -X ryu@<VM 주소>
```

사용자 이름은 ryu이고, 암호는 ryu입니다.

로그인 후, mn 명령으로 Mininet 환경을 시작합니다.

구축 환경은 호스트 3 대, 스위치 하나의 간단한 구성입니다.

mn 명령의 매개 변수는 다음과 같습니다.

매개변수	값	설명
topo	single,3	스위치 1 개, 호스트가 3 개인 토플로지
mac	없음	자동으로 호스트의 MAC 주소를 설정함
switch	ovsk	Open vSwitch를 사용
controller	remote	(별도로) 외부에서 OpenFlow 컨트롤러 사용
x	없음	xterm을 시작

실행 예는 다음과 같습니다.

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet>
```

실행하면 데스크탑 PC에서 5개의 xterm이 시작됩니다. 각 xterm은 호스트 1~3, 스위치, 그리고 컨트롤러에 대응합니다.

스위치에 대한 xterm에서 명령을 실행하여 사용하는 OpenFlow 버전을 설정합니다. 윈도우 제목이 「switch : s1 (root)」인 xterm이 스위치 용 xterm입니다.

우선 Open vSwitch의 상태를 확인합니다.

switch: s1:

```
root@ryu-vm:~# ovs-vsctl show
fdec0957-12b6-4417-9d02-847654e9cc1f
Bridge "s1"
    Controller "ptcp:6634"
    Controller "tcp:127.0.0.1:6633"
    fail_mode: secure
    Port "s1-eth3"
        Interface "s1-eth3"
    Port "s1-eth2"
        Interface "s1-eth2"
    Port "s1-eth1"
        Interface "s1-eth1"
    Port "s1"
        Interface "s1"
            type: internal
ovs_version: "1.11.0"
root@ryu-vm:~# ovs-dpctl show
system@ovs-system:
    lookups: hit:14 missed:14 lost:0
    flows: 0
    port 0: ovs-system (internal)
    port 1: s1 (internal)
    port 2: s1-eth1
    port 3: s1-eth2
    port 4: s1-eth3
root@ryu-vm:~#
```

스위치 (브리지) s1 이 생성되었고, 호스트에 해당 포트가 3개 추가되어 있습니다.

다음 OpenFlow 버전을 1.3으로 설정합니다.

switch: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
root@ryu-vm:~#
```

플로우 테이블을 확인해 봅시다.

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
root@ryu-vm:~#
```

ovs-ofctl 명령 실행시, 옵션으로 사용하는 OpenFlow 버전을 지정해야 합니다. 기본값은 OpenFlow10 입니다.

2.4.2 스위칭 허브 실행

모든 준비가 완료되었으므로, Ryu 응용 프로그램을 실행합니다.

윈도우 제목이 「controller : c0 (root)」인 xterm에서 다음 명령을 실행합니다.

controller: c0:

```
root@ryu-vm:~# ryu-manager --verbose ryu.app.simple_switch_13
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
instantiating app ryu.app.simple_switch_13
instantiating app ryu.controller.ofp_handler
BRICK SimpleSwitch13
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPPacketIn
```

```

BRICK ofp_event
    PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])}
    PROVIDES EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])}
    CONSUMES EventOFPErrorMsg
    CONSUMES EventOFPHello
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPswitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x2e2c050> address:(‘127.0.0.1’, 53937)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2e2a550>
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPswitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xff9ad15b OFPSwitchFeatures(auxiliary_id=0, capab...
move onto main mode

```

OVS와의 연결에 시간이 걸리는 경우도 있지만, 잠시 기다리면 다음과 같이

```

connected socket:<.....
hello ev ...
...
move onto main mode

```

로 표시됩니다.

이제 OVS와 연결되었고, 핸드쉐이크가 이루어져 Table-miss 흐름 항목이 추가되었고, 스위칭 허브는 Packet-In을 기다리는 상태입니다.

Table-miss 흐름 항목이 추가되어 있는지 확인합니다.

switch: s1:

```

root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=105.975s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER:65535
root@ryu-vm:~#

```

우선 순위가 0으로 매치가 없는 상태이고, 액션에 CONTROLLER 전송 데이터 크기로 65535 (0xffff = OF-PCML_NO_BUFFER)가 지정되어 있습니다.

2.4.3 동작 확인

호스트 1에서 호스트 2로 ping을 실행합니다.

1. ARP request

이 시점에서 호스트 1은 호스트 2의 MAC 주소를 모르기 때문에 ICMP echo request 이전에 ARP request를 브로드 캐스팅됩니다. 이 브로드 캐스트 패킷은 호스트 2 및 호스트 3에서 수신합니다.

2. ARP reply

호스트 2가 ARP에 응답하여 호스트 1에 ARP reply를 반환합니다.

3. ICMP echo request

이제 호스트 1 호스트 2의 MAC 주소를 알고 있으므로, echo request를 호스트 2에 보냅니다.

4. ICMP echo reply

호스트 2는 호스트 1의 MAC 주소를 이미 알고 있기 때문에, echo reply를 호스트 1에 반환합니다.

이렇게 통신이 이루어지는 것입니다.

ping 명령을 실행하기 전에 각 호스트에 어떤 패킷을 수신했는지 확인하기 위해 tcpdump 명령을 실행합니다.

host: h1:

```
root@ryu-vm:~# tcpdump -en -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

host: h2:

```
root@ryu-vm:~# tcpdump -en -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

host: h3:

```
root@ryu-vm:~# tcpdump -en -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

그럼, 먼저 mn 명령을 실행한 콘솔에서 다음 명령을 실행하여 호스트 1에서 호스트 2로 ping을 수행합니다.

```
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=97.5 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 97.594/97.594/97.594/0.000 ms
mininet>
```

ICMP echo reply가 정상적으로 반환됩니다.

우선, 플로우 테이블을 확인합니다.

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=417.838s, table=0, n_packets=3, n_bytes=182, priority=0 actions=CONTROLLER:0
  cookie=0x0, duration=48.444s, table=0, n_packets=2, n_bytes=140, priority=1,in_port=2,dl_dst=00:00:00:00:00:00
  cookie=0x0, duration=48.402s, table=0, n_packets=1, n_bytes=42, priority=1,in_port=1,dl_dst=00:00:00:00:00:00
root@ryu-vm:~#
```

Table-miss 플로우 항목 이외에 우선 순위가 1인 플로우 항목이 2 개 등록되어 있습니다.

1. 수신 포트 (in_port):2, MAC 수신 주소(dl_dst):호스트 1 → 동작(actions):포트1로 전송
2. 수신 포트 (in_port):1, MAC 수신 주소(dl_dst):호스트 2 → 동작(actions):포트2로 전송

(1) 항목은 2 번 reference되고 (n_packets), (2) 항목은 1 번 reference됩니다. (1)은 호스트 2에서 호스트 1로의 통신이므로, ARP reply 및 ICMP echo reply 두 가지에 일치해야 합니다. (2)는 호스트 1에서 호스트 2로의 통신에서, ARP request가 브로드캐스트되므로 이는 ICMP echo request에 의한 것입니다.

그럼 simple_switch_13 로그 결과를 살펴 봅시다.

controller: c0:

```
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:01 00:00:00:00:00:02 1
```

첫 번째 Packet-In은 호스트 1이 발행한 ARP request이고, 브로드캐스트이므로 플로우 항목에 등록되지 않고 Packet-Out 만 발행됩니다.

두 번째는 호스트 2에서 반환된 ARP reply에서 목적지 MAC 주소가 호스트 1이고 따라서 위의 플로우 항목 (1)이 등록됩니다.

세 번째는 호스트 1에서 호스트 2로 전송 된 ICMP echo request에서 플로우 항목 (2)이 등록됩니다.

호스트 2에서 호스트 1에 반환 된 ICMP echo reply는 등록된 플로우 항목 (1)에 일치하기 때문에 Packet-In은 발행되지 않고 호스트 1에 전송됩니다.

마지막으로 각 호스트에서 실행한 tcpdump의 출력 결과를 살펴봅시다.

host: h1:

```
root@ryu-vm:~# tcpdump -en -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
20:38:04.625473 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Request
20:38:04.678698 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42: Reply 1
20:38:04.678731 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800), length 98: 10.0.0.1
20:38:04.722973 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800), length 98: 10.0.0.1
```

호스트 1에서 먼저 ARP request가 브로드캐스트되고 있어, 계속 호스트 2에서 반환된 ARP reply를 받고 있습니다. 그런 다음, 호스트 1은 ICMP echo request를 발행하고, 호스트 2에서 반환된 ICMP echo reply를 수신합니다.

host: h2:

```
root@ryu-vm:~# tcpdump -en -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
20:38:04.637987 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Request
20:38:04.638059 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42: Reply 1
20:38:04.722601 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800), length 98: 10.0.0.2
20:38:04.722747 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800), length 98: 10.0.0.2
```

호스트 2에서 호스트 1이 발행한 ARP request를 수신하고, 호스트 1에 ARP reply를 반환합니다. 그런 다음, 호스트 1에서 ICMP echo request를 수신하고 호스트 1에 echo reply를 반환합니다.

host: h3:

```
root@ryu-vm:~# tcpdump -en -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
20:38:04.637954 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Request
```

호스트 3은 먼저 호스트 1의 브로드캐스팅 된 ARP request 만 수신 하고 있습니다.

2.5 정리

이 장에서는 간단한 스위칭 허브 구현을 주제로 Ryu 응용 프로그램 구현에 대해 기본적인 절차와 OpenFlow에 따른 OpenFlow 스위치의 간단한 제어 방법을 설명하였습니다.

트래픽 모니터

이 장에서는 「스위칭 허브」에서 다루었던 스위칭 허브에 OpenFlow 스위치 통계 정보를 모니터링하는 기능을 추가하는 법에 대해 살펴봅니다.

3.1 네트워크 정기 검사

네트워크는 이미 많은 서비스 및 비즈니스 인프라가 있기 때문에 정상 상태 및 안정적인 가동이 유지되기 를 요구합니다. 그러나, 항상 뭔가 문제 가 발생합니다.

네트워크에 이상이 발생했을 경우, 신속하게 원인을 파악하고 복구시켜야 됩니다. 말할 것도 없이, 이상을 감지하고 원인을 파악하기 위해서는 평소부터 네트워크의 상태를 잘 이해할 필요가 있습니다. 예를 들어, 네트워크 장비 내 포트에서 트래픽 양이 매우 높은 값을 보여주는 경우, 그것이 비정상적인 상태인지 정상 상태인지, 또는 언제부터 이렇게 되었는가하는 것은, 지속적으로 해당 포트의 트래픽 양을 측정하지 않으면 판단할 수 없습니다.

그래서, 네트워크의 건강 상태를 지속적으로 모니터링하고 계속하는 것은, 해당 네트워크를 사용하는 서비스와 업무의 지속적인 안정적 운용을 위해서도 필수입니다. 물론, 트래픽 정보를 단순히 모니터링한다고 해서 완벽한 보장을 하지는 않습니다. 하지만, 이 장에서는 OpenFlow를 사용해 스위치의 통계 정보를 얻는 방법에 대해 설명하고자 합니다.

3.2 트래픽 모니터 구현

다음은 「스위칭 허브」에서 설명한 스위칭 허브에 트래픽 모니터 기능을 추가한 소스 코드입니다.

```
from operator import attrgetter

from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub

class SimpleMonitor(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)
```

```

@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in self.datapaths:
            self.logger.debug('register datapath: %016x', datapath.id)
            self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('unregister datapath: %016x', datapath.id)
            del self.datapaths[datapath.id]

def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
        hub.sleep(10)

def _request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)

    req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
    datapath.send_msg(req)

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    body = ev.msg.body

    self.logger.info('datapath         '
                     'in-port eth-dst           '
                     'out-port packets  bytes')
    self.logger.info('-----  '
                     '-----   -----  ')
    for stat in sorted([flow for flow in body if flow.priority == 1],
                       key=lambda flow: (flow.match['in_port'],
                                         flow.match['eth_dst'])):
        self.logger.info(' %016x %8x %17s %8x %8d %8d',
                        ev.msg.datapath.id,
                        stat.match['in_port'], stat.match['eth_dst'],
                        stat.instructions[0].actions[0].port,
                        stat.packet_count, stat.byte_count)

@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    body = ev.msg.body

    self.logger.info('datapath      port      '
                     'rx-pkts rx-bytes rx-error '
                     'tx-pkts tx-bytes tx-error')
    self.logger.info('-----  -----  ')
    for stat in sorted(body, key=attrgetter('port_no')):
        self.logger.info(' %016x %8x %8d %8d %8d %8d',
                        ev.msg.datapath.id, stat.port_no,
                        stat.rx_packets, stat.rx_bytes, stat.rx_errors,

```

```
stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```

SimpleSwitch13을 상속하는 SimpleMonitor 클래스에 트래픽 모니터링 기능을 구현하고 있기 때문에, 여기에는 패킷 전송에 대한 처리 부분이 없습니다.

3.2.1 고정 주기 처리

스위칭 허브의 처리와 병행하여 주기적으로 통계 정보를 얻기 위해 OpenFlow 스위치에 요청하는 스레드를 생성합니다.

```
from operator import attrgetter

from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub

class SimpleMonitor(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)
# ...
```

ryu.lib.hub에는 몇 가지 eventlet wrapper와 기본 클래스 구현이 있습니다. 여기에서는 스레드를 생성하는 hub.spawn () 을 사용합니다. 실제로 생성되는 스레드는 eventlet green thread입니다.

```
# ...
@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in self.datapaths:
            self.logger.debug('register datapath: %016x', datapath.id)
            self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('unregister datapath: %016x', datapath.id)
            del self.datapaths[datapath.id]

def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
        hub.sleep(10)
# ...
```

스레드 함수 _monitor () 에서 등록된 스위치에 대한 통계 가져오기 요청을 10 초 간격으로 무한 반복 합니다.

연결된 스위치를 모니터링하기 때문에 스위치의 접속 및 접속 끊김에 대한 EventOFPStateChange 이벤트를 이용하고 있습니다. 이 이벤트는 Ryu 프레임 워크가 발행하는 것으로, Datapath의 상태가 바뀌었을 때에 발행됩니다.

여기에서는 Datapath 상태가 MAIN_DISPATCHER 가 될 때, 해당 스위치는 모니터링 대상으로 등록되고, DEAD_DISPATCHER 가 될 때, 등록이 삭제됩니다.

```
# ...
def _request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)

    req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
    datapath.send_msg(req)
# ...
```

주기적으로 호출되는 `_request_stats()`는 스위치에 `OFPFlowStatsRequest` 와 `OFPPortStatsRequest`를 발행하고 있습니다.

`OFPFlowStatsRequest`는 플로우 항목에 대한 통계를 스위치에 요청합니다. 테이블 ID, 출력 포트, cookie 값, 매치 등의 상태를 통해 요청 대상 플로우 항목을 좁힐 수 있지만, 여기에서는 모든 플로우 항목을 대상으로 하고 있습니다.

`OFPPortStatsRequest`는 포트 관련 통계 정보를 스위치에 요청합니다. 원하는 포트 번호를 정보 수집을 위해 지정할 수 있습니다. 여기에서는 `OFPP_ANY`를 지정하여 모든 포트의 통계 정보를 요청하고 있습니다.

3.2.2 FlowStats

스위치로부터 응답을 받기 위해 `FlowStatsReply` 메시지를 수신하는 이벤트 처리기를 생성합니다.

```
# ...
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    body = ev.msg.body

    self.logger.info('datapath         '
                     'in-port eth-dst           '
                     'out-port packets bytes')
    self.logger.info('-----  '
                     '-----   -----  ')
    for stat in sorted([flow for flow in body if flow.priority == 1],
                       key=lambda flow: (flow.match['in_port'],
                                         flow.match['eth_dst'])):
        self.logger.info('%016x %8x %17s %8x %8d %8d',
                         ev.msg.datapath.id,
                         stat.match['in_port'], stat.match['eth_dst'],
                         stat.instructions[0].actions[0].port,
                         stat.packet_count, stat.byte_count)
# ...
```

`OPFFlowStatsReply` 클래스의 속성인 `body`는 `OFPFlowStats` 목록에서 `FlowStatsRequest`의 대상이 된 각 플로우 항목의 통계 정보가 포함되어 있습니다.

우선 순위가 0인 Table-miss 플로우를 제외하고 모든 플로우 항목을 선택합니다. 수신 포트와 대상 MAC 주소로 정렬하여 각각의 플로우 항목과 매치되는 패킷과 바이트를 출력합니다.

또한, 여기에서는 일부 숫자들만 로그에 출력되고 있지만, 지속적으로 정보를 수집하고 분석하려면 외부 프로그램과의 연계가 필요할 것입니다. 그런 경우 `OPFFlowStatsReply`의 내용을 JSON 형식으로 변환할 수 있습니다.

예를 들어 다음과 같이 쓸 수 있습니다.

```
import json
```

```
# ...
self.logger.info('%s', json.dumps(ev.msg.to_jsondict(), ensure_ascii=True,
                                   indent=3, sort_keys=True))
```

이 경우 다음과 같이 출력됩니다.

```
{
    "OFPFlowStatsReply": {
        "body": [
            {
                "OFPFlowStats": {
                    "byte_count": 0,
                    "cookie": 0,
                    "duration_nsec": 680000000,
                    "duration_sec": 4,
                    "flags": 0,
                    "hard_timeout": 0,
                    "idle_timeout": 0,
                    "instructions": [
                        {
                            "OFPInstructionActions": {
                                "actions": [
                                    {
                                        "OFPActionOutput": {
                                            "len": 16,
                                            "max_len": 65535,
                                            "port": 4294967293,
                                            "type": 0
                                        }
                                    }
                                ],
                                "len": 24,
                                "type": 4
                            }
                        }
                    ],
                    "length": 80,
                    "match": {
                        "OFPMatch": {
                            "length": 4,
                            "oxm_fields": [],
                            "type": 1
                        }
                    },
                    "packet_count": 0,
                    "priority": 0,
                    "table_id": 0
                }
            },
            {
                "OFPFlowStats": {
                    "byte_count": 42,
                    "cookie": 0,
                    "duration_nsec": 72000000,
                    "duration_sec": 57,
                    "flags": 0,
                    "hard_timeout": 0,
                    "idle_timeout": 0,
                    "instructions": [
                        {
                            "OFPInstructionActions": {
                                "actions": [

```

```

        {
            "OFPActionOutput": {
                "len": 16,
                "max_len": 65509,
                "port": 1,
                "type": 0
            }
        }
    ],
    "len": 24,
    "type": 4
}
],
"length": 96,
"match": [
    "OFPMatch": {
        "length": 22,
        "oxm_fields": [
            {
                "OXMTlv": {
                    "field": "in_port",
                    "mask": null,
                    "value": 2
                }
            },
            {
                "OXMTlv": {
                    "field": "eth_dst",
                    "mask": null,
                    "value": "00:00:00:00:00:01"
                }
            }
        ],
        "type": 1
    },
    "packet_count": 1,
    "priority": 1,
    "table_id": 0
}
},
"flags": 0,
"type": 1
}
}
}

```

3.2.3 PortStats

스위치로부터 응답을 받기 위해 PortStatsReply 메시지를 수신하는 이벤트 처리기를 생성합니다.

```

# ...
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    body = ev.msg.body

    self.logger.info('datapath      port      '
                     'rx-pkts  rx-bytes rx-error '
                     'tx-pkts  tx-bytes tx-error')
    self.logger.info('----- ----- '

```

```

        '-----')
for stat in sorted(body, key=attrgetter('port_no')):
    self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',
                      ev.msg.datapath.id, stat.port_no,
                      stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                      stat.tx_packets, stat.tx_bytes, stat.tx_errors)

```

OPFPortStatsReply 클래스의 속성 body 은 OFPPortStats 의 목록에 있습니다.

OFPPortStats 에는 포트 번호, 송수신 각각의 패킷 수, 바이트 수, 드롭 개수, 오류 개수, 프레임 오류 개수, 오버런 개수, CRC 오류 개수, 충돌 개수 등 통계 정보가 저장됩니다.

여기에서는 포트 번호별로 정렬하고 수신 패킷 개수, 수신된 바이트 수, 송신 오류 개수, 전송 패킷 수, 송신 바이트 수, 전송 오류 개수를 출력합니다.

3.3 트래픽 모니터 실행

그럼 실제로 이 트래픽 모니터를 실행 해 봅시다.

먼저 「[스위칭 허브](#)」와 같이 Mininet을 실행합니다. 여기서 스위치 OpenFlow 버전에 OpenFlow13을 설정하는 것을 잊지 마십시오.

다음, 이제, 트래픽 모니터를 실행합니다.

controller: c0:

```

ryu@ryu-vm:~# ryu-manager --verbose ./simple_monitor.py
loading app ./simple_monitor.py
loading app ryu.controller.ofp_handler
instantiating app ./simple_monitor.py
instantiating app ryu.controller.ofp_handler
BRICK SimpleMonitor
    CONSUMES EventOFPStateChange
    CONSUMES EventOFPFlowStatsReply
    CONSUMES EventOFPPortStatsReply
    CONSUMES EventOFPacketIn
    CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
    PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
    PROVIDES EventOFPFlowStatsReply TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPPortStatsReply TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPacketIn TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPSwitchFeatures TO {'SimpleMonitor': set(['config'])}
    CONSUMES EventOFPErrorMsg
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPHello
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x343fb10> address:(‘127.0.0.1’, 55598)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x343fed0>
move onto config mode
EVENT ofp_event->SimpleMonitor EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x7dd2dc58 OFPSwitchFeatures(auxiliary_id=0, capab
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000001
send stats request: 0000000000000001
EVENT ofp_event->SimpleMonitor EventOFPFlowStatsReply
datapath      in-port   eth-dst          out-port  packets  bytes
-----  -----  -----  -----  -----
EVENT ofp_event->SimpleMonitor EventOFPPortStatsReply

```

datapath	port	rx-pkts	rx-bytes	rx-error	tx-pkts	tx-bytes	tx-error
00000000000000000001	1	0	0	0	0	0	0
00000000000000000001	2	0	0	0	0	0	0
00000000000000000001	3	0	0	0	0	0	0
00000000000000000001	ffffffe	0	0	0	0	0	0

「[스위칭 허브](#)」에서는 ryu-manager 명령에 SimpleSwitch13 모듈 이름 (ryu.app.simple_switch_13)을 지정했지만, 여기에서는 SimpleMonitor의 파일 이름 ([./simple_monitor.py](#))을 지정합니다.

여기서는, 플로우 항목이 없고, (Table-miss 플로우 항목은 표시되지 않습니다) 각 포트의 개수도 모두 0입니다.

호스트 1에서 호스트 2로 ping을 실행하자.

host: h1:

```
root@ryu-vm:~# ping -c1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=94.4 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 94.489/94.489/94.489/0.000 ms
root@ryu-vm:~#
```

패킷 전송 및 플로우 항목이 등록되고 통계 정보가 변경됩니다.

controller: c0:

datapath	in-port	eth-dst		out-port	packets	bytes	
00000000000000000001	1	00:00:00:00:00:02		2	1	42	
00000000000000000001	2	00:00:00:00:00:01		1	2	140	
datapath	port	rx-pkts	rx-bytes	rx-error	tx-pkts	tx-bytes	tx-error
00000000000000000001	1	3	182	0	3	182	0
00000000000000000001	2	3	182	0	3	182	0
00000000000000000001	3	0	0	0	1	42	0
00000000000000000001	ffffffe	0	0	0	1	42	0

플로우 항목의 통계 정보에 따르면, 수신 포트 1의 플로우에 매치된 트래픽은 1 패킷, 42 바이트라고 기록되어 있습니다. 수신 포트 2에서는 2 패킷, 140 바이트로 기록되어 있습니다.

포트 통계 정보에 따르면, 포트 1의 수신 패킷 수 (rx-pkts)는 3, 수신 바이트 수 (rx-bytes)는 182 바이트, 포트 2는 3 패킷, 182 바이트라고 되어 있습니다.

플로우 항목의 통계 정보와 해당 포트의 통계 화면이 일치하지는 않습니다. 이유는 플로우 항목의 통계 정보가 해당 항목에 매치되고 전송된 패킷 정보이기 때문입니다. 즉, Table-miss에 의해 Packet-In을 발행되어 Packet-Out으로 전송된 패킷은 해당 통계의 대상에 포함되지 않기 때문입니다.

이 경우, 호스트 1이 먼저 브로드 캐스트한 ARP 요청, 호스트 2가 호스트 1에 반환하는 ARP 응답, 그리고 호스트 1에서 호스트 2로 발행하는 echo 요청인 3개의 패킷이 있고, (모두) Packet-Out 의해 전송됩니다. 이러한 이유로, 포트 통계치는 플로우 항목의 통계치보다 많습니다.

3.4 정리

이 장에서는 통계 정보 수집 기능을 주제로 하여, 다음 항목들을 설명하였습니다.

- Ryu 응용 프로그램에서의 스레드 생성 방법
- Datapath의 상태 변화 확인
- FlowStats 및 PortStats 수집 방법

REST연계

이 장에서는 「[스위칭 허브](#)」에서 설명한 스위칭 허브에 REST 연계 기능을 추가합니다.

4.1 REST API의 기본

Ryu에는 WSGI에 대응 한 Web 서버의 기능이 있습니다. 이 기능을 이용하여 다른 시스템이나 브라우저 등과의 연계를 할 때 유용한, REST API를 만들 수 있습니다.

주석: WSGI는 Python에서 Web 어플리케이션과 Web 서버 연결을 위한 통일 된 프레임워크를 의미합니다.

4.2 REST API된 스위칭 허브 구현

「[스위칭 허브](#)」에서 설명한 스위칭 허브에 다음 두 REST API를 추가하여 봅시다.

1. MAC 주소 테이블 취득 API

스위칭 허브가 보유하고 있는 MAC 주소 테이블의 내용을 반환합니다. MAC 주소와 포트 번호 쌍을 JSON 형식으로 반환합니다.

2. MAC 주소 테이블 등록 API

MAC 주소와 포트 번호 쌍을 MAC 주소 테이블에 등록하고 스위치 흐름 항목의 추가를 합니다.

그려면 소스 코드를 살펴 봅시다.

```
import json
import logging

from ryu.app import simple_switch_13
from webob import Response
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.app.wsgi import ControllerBase, WSGIApplication, route
from ryu.lib import dpid as dpid_lib

simple_switch_instance_name = 'simple_switch_api_app'
url = '/simpleswitch/mactable/{dpid}'

class SimpleSwitchRest13(ControllerBase):
    def __init__(self, app, **config):
        super(SimpleSwitchRest13, self).__init__(app, simple_switch_instance_name, url, **config)
        self.app = app
```

```

_CONTEXTS = { 'wsgi': WSGIApplication }

def __init__(self, *args, **kwargs):
    super(SimpleSwitchRest13, self).__init__(*args, **kwargs)
    self.switches = {}
    wsgi = kwargs['wsgi']
    wsgi.register(SimpleSwitchController, {simple_switch_instance_name : self})

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    super(SimpleSwitchRest13, self).switch_features_handler(ev)
    datapath = ev.msg.datapath
    self.switches[datapath.id] = datapath
    self.mac_to_port.setdefault(datapath.id, {})

def set_mac_to_port(self, dpid, entry):
    mac_table = self.mac_to_port.setdefault(dpid, {})
    datapath = self.switches.get(dpid)

    entry_port = entry['port']
    entry_mac = entry['mac']

    if datapath is not None:
        parser = datapath.ofproto_parser
        if entry_port not in mac_table.values():

            for mac, port in mac_table.items():

                # from known device to new device
                actions = [parser.OFPActionOutput(entry_port)]
                match = parser.OFPMatch(in_port=port, eth_dst=entry_mac)
                self.add_flow(datapath, 1, match, actions)

                # from new device to known device
                actions = [parser.OFPActionOutput(port)]
                match = parser.OFPMatch(in_port=entry_port, eth_dst=mac)
                self.add_flow(datapath, 1, match, actions)

            mac_table.update({entry_mac : entry_port})
    return mac_table

class SimpleSwitchController(ControllerBase):

    def __init__(self, req, link, data, **config):
        super(SimpleSwitchController, self).__init__(req, link, data, **config)
        self.simple_switch_spp = data[simple_switch_instance_name]

    @route('simpleswitch', url, methods=['GET'], requirements={'dpid': dpid_lib.DPID_PATTERN})
    def list_mac_table(self, req, **kwargs):

        simple_switch = self.simple_switch_spp
        dpid = dpid_lib.str_to_dpid(kwargs['dpid'])

        if dpid not in simple_switch.mac_to_port:
            return Response(status=404)

        mac_table = simple_switch.mac_to_port.get(dpid, {})
        body = json.dumps(mac_table)
        return Response(content_type='application/json', body=body)

    @route('simpleswitch', url, methods=['PUT'], requirements={'dpid': dpid_lib.DPID_PATTERN})
    def put_mac_table(self, req, **kwargs):

```

```

simple_switch = self.simple_switch_spp
dpid = dpid_lib.str_to_dpid(kwargs['dpid'])
new_entry = eval(req.body)

if dpid not in simple_switch.mac_to_port:
    return Response(status=404)

try:
    mac_table = simple_switch.set_mac_to_port(dpid, new_entry)
    body = json.dumps(mac_table)
    return Response(content_type='application/json', body=body)
except Exception as e:
    return Response(status=500)

```

simple_switch_rest_13.py에서는 두 클래스를 정의하고 있습니다.

첫째, HTTP 요청을 받는 URL과 해당 메서드를 정의하는 컨트롤러 SimpleSwitchController 입니다.
두 번째는 「 스위칭 허브 」를 확장하고 MAC 주소 테이블 업데이트를 할 수 있도록 한 클래스 SimpleSwitchRest13 입니다.

SimpleSwitchRest13 는 스위치에 흐름 항목을 추가하기 위해 FeaturesReply 메서드를 오버라이드하고 datapath 객체를 보유하고 있습니다.

4.3 SimpleSwitchRest13 클래스의 구현

```

class SimpleSwitchRest13(simple_switch_13.SimpleSwitch13):

    _CONTEXTS = { 'wsgi': WSGIApplication }
    ...

```

클래스 변수 _CONTEXTS에서 Ryu의 WSGI 대응 Web 서버의 클래스를 지정합니다. 그러면 wsgi라는 키에서 WSGI의 Web 서버 인스턴스가 얻을 수 있습니다.

```

def __init__(self, *args, **kwargs):
    super(SimpleSwitchRest13, self).__init__(*args, **kwargs)
    self.switches = {}
    wsgi = kwargs['wsgi']
    wsgi.register(SimpleSwitchController, {simple_switch_instance_name : self})
    ...

```

생성자는 후술하는 컨트롤러 클래스를 등록하기 위하여, WSGIApplication의 인스턴스를 취득하고 있습니다. 등록은 register 메서드를 사용합니다. register 메소드 실행시 컨트롤러의 생성자에서 SimpleSwitchRest13 클래스의 인스턴스에 액세스 할 수 있도록 simple_switch_api_app라는 키 이름 사전 개체를 전달합니다.

```

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    super(SimpleSwitchRest13, self).switch_features_handler(ev)
    datapath = ev.msg.datapath
    self.switches[datapath.id] = datapath
    self.mac_to_port.setdefault(datapath.id, {})
    ...

```

부모 클래스의 switch_features_handler 을 무시하고 있습니다. 이 메서드는 SwitchFeatures 이벤트가 발생한 시간에 이벤트 객체 ev에 포함 된 datapath 개체를 가져옵니다. 인스턴스 변수 switches 보유하고 있습니다. 또한 이 시기에 MAC 주소 테이블에 기본값으로 빈 사전을 설정하고 있습니다.

```

def set_mac_to_port(self, dpid, entry):
    mac_table = self.mac_to_port.setdefault(dpid, {})
    datapath = self.switches.get(dpid)

```

```

entry_port = entry['port']
entry_mac = entry['mac']

if datapath is not None:
    parser = datapath.ofproto_parser
    if entry_port not in mac_table.values():

        for mac, port in mac_table.items():

            # from known device to new device
            actions = [parser.OFPActionOutput(entry_port)]
            match = parser.OFPMatch(in_port=port, eth_dst=entry_mac)
            self.add_flow(datapath, 1, match, actions)

            # from new device to known device
            actions = [parser.OFPActionOutput(port)]
            match = parser.OFPMatch(in_port=entry_port, eth_dst=mac)
            self.add_flow(datapath, 1, match, actions)

        mac_table.update({entry_mac : entry_port})
    return mac_table
...

```

지정된 스위치에 MAC 주소와 포트를 등록하는 메서드입니다. REST API가 PUT 방식으로 불리는이 실행됩니다.

인수 entry에는 등록을하려는 MAC 주소와 연결 포트 쌍이 포함되어 있습니다.

MAC 주소 테이블 self.mac_to_port의 정보를 참조하여 스위치에 등록하는 흐름 항목을 찾아갑니다.

예를 들어, MAC 주소 테이블에 다음의 MAC 주소와 연결 포트 쌍이 등록되어 있고,

- 00:00:00:00:00:01, 1

인수 entry에 전달 된 MAC 주소와 포트 쌍이

- 00:00:00:00:00:02, 2

해당 스위치에 등록해야하는 흐름 항목은 다음과 같습니다.

- 매칭 조건 : in_port = 1, dst_mac = 00:00:00:00:00:02 조치 : output = 2
- 매칭 조건 : in_port = 2, dst_mac = 00:00:00:00:00:01 조치 : output = 1

흐름 항목의 등록은 부모 클래스의 add_flow 메소드를 이용하고 있습니다. 마지막으로, 인수 entry에서 전달 된 정보를 MAC 주소 테이블에 저장합니다.

4.4 SimpleSwitchController 클래스의 구현

다음은 REST API에 대한 HTTP 요청을 수락 컨트롤러 클래스입니다. 클래스 이름은 SimpleSwitchController입니다.

```

class SimpleSwitchController(ControllerBase):
    def __init__(self, req, link, data, **config):
        super(SimpleSwitchController, self).__init__(req, link, data, **config)
        self.simple_switch_spp = data[simple_switch_instance_name]
    ...

```

생성자에서 SimpleSwitchRest13 클래스의 인스턴스를 가져옵니다.

```

@route('simpleswitch', url, methods=['GET'], requirements={'dpid': dpid_lib.DPID_PATTERN})
def list_mac_table(self, req, **kwargs):

    simple_switch = self.simple_switch_spp
    dpid = dpid_lib.str_to_dpid(kwargs['dpid'])

```

```

if dpid not in simple_switch.mac_to_port:
    return Response(status=404)

mac_table = simple_switch.mac_to_port.get(dpid, {})
body = json.dumps(mac_table)
return Response(content_type='application/json', body=body)
...

```

REST API URL과 해당 프로세스를 구현하는 부분입니다. 이 방법과 URL와의 바인딩이 Ryu에서 정의 된 route 장식을 이용하고 있습니다.

장식으로 지정하는 내용은 다음과 같습니다.

- 제1인수

모든 이름

- 제2인수

URL을 지정합니다. URL이 `http://<서버 IP>:8080/simpleswitch/mactable/<데이터 ID>` 가 되도록 합니다.

- 제3인수

HTTP 메서드를 지정합니다. GET 메서드를 지정합니다.

- 제4인수

지정 위치의 형식을 지정합니다. URL(/simpleswitch/mactable/{dpid})의 {dpid} 부분이 ryu/lib/dpid.py의 DPID_PATTERN에서 정의 된 16 자리의 16 진수 표현에 따르는 것을 조건으로하고 있습니다.

제 2 인수로 지정된 URL에서 REST API가 불려 그 때의 HTTP 메소드가 GET 인 경우 list_mac_table 메소드를 호출합니다. 이 메소드는, {dpid} 부분에서 지정된 데이터 경로 ID에 해당하는 MAC 주소 테이블을 검색하고 JSON으로 변환 호출자에게 반환합니다.

또한, Ryu에 연결하지 않은 미지의 스위치의 데이터 경로 ID를 지정하면 응답 코드 404을 반환합니다.

```

@route('simpleswitch', url, methods=['PUT'], requirements={'dpid': dpid_lib.DPID_PATTERN})
def put_mac_table(self, req, **kwargs):

    simple_switch = self.simple_switch_spp
    dpid = dpid_lib.str_to_dpid(kwargs['dpid'])
    new_entry = eval(req.body)

    if dpid not in simple_switch.mac_to_port:
        return Response(status=404)

    try:
        mac_table = simple_switch.set_mac_to_port(dpid, new_entry)
        body = json.dumps(mac_table)
        return Response(content_type='application/json', body=body)
    except Exception as e:
        return Response(status=500)
...

```

다음은 MAC 주소 테이블을 등록하는 REST API입니다.

URL은 MAC 주소 테이블 취득시의 API와 동일하지만 HTTP 메서드가 PUT의 경우 put_mac_table 메소드를 호출합니다. 이 메서드는 내부적으로 스위칭 허브 인스턴스의 set_mac_to_port 메서드를 호출합니다. 또한 put_mac_table 메소드 내에서 예외가 발생하면 응답 코드 500을 반환합니다. 또한 list_mac_table 메소드뿐만 아니라 Ryu에 연결하지 않은 미지의 스위치의 데이터 경로 ID를 지정하면 응답 코드 404을 반환합니다.

4.5 REST API 탑재 스위칭 허브의 실행

REST API를 추가 한 스위칭 허브를 실행하자.

먼저 「스위칭 허브」와 같이 Mininet를 실행합니다. 여기서도 스위치 OpenFlow 버전에 OpenFlow13을 설정하는 것을 잊지 마십시오. 이어 REST API를 추가 한 스위칭 허브를 시작합니다.

```
ryu@ryu-vm:~/ryu/ryu/app$ cd ~/ryu/ryu/app
ryu@ryu-vm:~/ryu/ryu/app$ sudo ovs-vsctl set Bridge s1 protocols=OpenFlow13
ryu@ryu-vm:~/ryu/ryu/app$ ryu-manager --verbose ./simple_switch_rest_13.py
loading app ./simple_switch_rest_13.py
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app ryu.controller.ofp_handler
instantiating app ./simple_switch_rest_13.py
BRICK SimpleSwitchRest13
    CONSUMES EventOFPPacketIn
    CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
    PROVIDES EventOFPPacketIn TO {'SimpleSwitchRest13': set(['main'])}
    PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitchRest13': set(['config'])}
    CONSUMES EventOFPErrorMsg
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPHello
(31135) wsgi starting up on http://0.0.0.0:8080/
connected socket:<eventlet.greenio.GreenSocket object at 0x318c6d0> address:(‘127.0.0.1’, 48914)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x318cc10>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x78dd7a72 OFPSwitchFeatures(auxiliary_id=0, capab
move onto main mode
```

시작 메시지에 “(31135) wsgi starting up on <http://0.0.0.0:8080/>” 줄이 있습니다만, 이것은 Web 서버가 포트 번호 8080으로 시작했음을 나타냅니다.

다음 mininet 쉘에서 h1에서 h2로 ping을 실행합니다.

```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=84.1 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 84.171/84.171/84.171/0.000 ms
```

이때 Ryu의 Packet-In은 3회 발생하고 있습니다.

```
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
```

여기서, 스위칭 허브의 MAC 테이블을 검색하는 REST API를 실행하자. 이번에는 REST API 호출에 curl 명령을 사용합니다.

```
ryu@ryu-vm:~$ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/00000000000000000000000000000000
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

h1과 h2 두 호스트가 MAC 주소 테이블에서 학습 된 것을 알 수 있습니다.

이번에는 h1, h2의 두 호스트를 미리 MAC 주소 테이블에 저장하고 ping을 실행 해 봅니다. 일단 스위칭 허브와 Mininet을 중지합니다. 그런 다음 다시 Mininet를 시작하고 OpenFlow 버전을 OpenFlow13로 설정 후 스위칭 허브를 시작합니다.

...

```
(26759) wsgi starting up on http://0.0.0.0:8080/
connected socket:<eventlet.greenio.GreenSocket object at 0x2afe6d0> address:(‘127.0.0.1’, 48818)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2afec10>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPswitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x96681337 OFPSwitchFeatures(auxiliary_id=0, capable
switch_features_handler inside sub class
move onto main mode
```

그런 다음 MAC 주소 테이블 업데이트를 위한 REST API를 1 호스트마다 호출합니다. REST API를 호출 할 때 데이터 형식은 { “mac”: “MAC 주소”, “port”: 연결 포트 번호}가 되도록 합니다.

```
ryu@ryu-vm:~$ curl -X PUT -d ‘{"mac" : "00:00:00:00:00:01", "port" : 1}' http://127.0.0.1:8080/simplest-switch/mactable/00:00:00:00:00:01
ryu@ryu-vm:~$ curl -X PUT -d ‘{"mac" : "00:00:00:00:00:02", "port" : 2}' http://127.0.0.1:8080/simplest-switch/mactable/00:00:00:00:00:02
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

이 명령을 실행하면 h1, h2에 대응 한 흐름 항목이 스위치에 등록됩니다.

이어 h1에서 h2로 ping을 실행합니다.

```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=4.62 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.623/4.623/4.623/0.000 ms

...
move onto main mode
(28293) accepted ('127.0.0.1', 44453)
127.0.0.1 - - [19/Nov/2013 19:59:45] "PUT /simpleswitch/mactable/0000000000000001 HTTP/1.1" 200 11
EVENT ofp_event->SimpleSwitchRest13 EventOFPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
```

이때 스위치는 이미 흐름 항목이 존재하기 때문에 Packet-In은 h1에서 h2의 ARP 요청 때만 발생 이후의 패킷 교환에서는 발생하지 않습니다.

4.6 정리

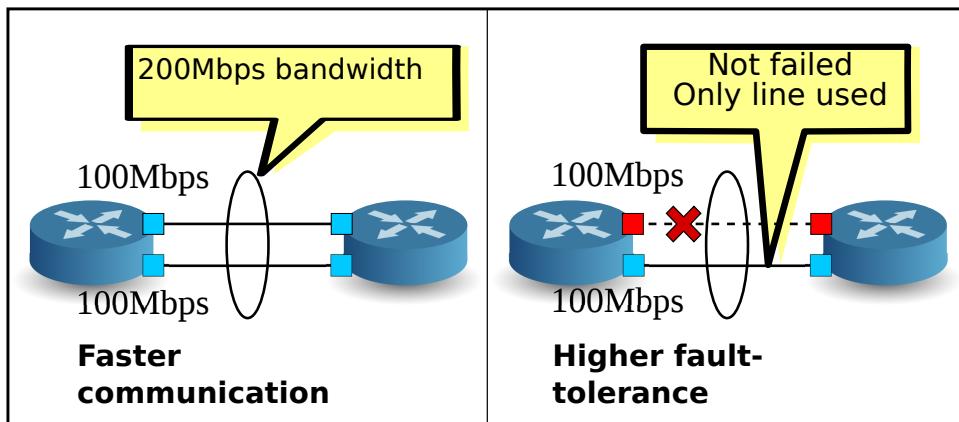
이 장에서는 MAC 주소 테이블 참조하거나 업데이트하는 기능을 소재로 REST API를 추가하는 방법에 대해 설명했습니다. 기타 응용으로서 스위치에 어떤 흐름 항목을 추가 할 수 같은 REST API를 만들고 브라우저에서 사용할 수 있도록 하는 것도 좋은 것이 아닐까요.

링크 어그리게이션

이 장에서는 Ryu를 이용한 링크 어그리게이션 기능을 구현하는 방법을 설명하여 마련했습니다.

5.1 링크 어그리게이션

링크 어그리게이션은 IEEE802.1AX-2008에서 규정 된 여러 물리적 회선을 묶어 하나의 논리적 링크로 처리 기술입니다. 링크 통합 기능은 특정 네트워크 장치 간의 통신 속도를 향상시킬 수 있으며 동시에 에 중복성을 확보함으로써 감수성을 향상시킬 수 있습니다.



링크 통합 기능을 사용하려면 각 네트워크 장비에서 어떤 인터페이스를 어떤 그룹으로 묶을 것인가하는 설정을 미리 해두 해야합니다.

링크 통합 기능을 시작하는 방법에는 각각의 네트워크 장비에 대해 직접 지시 할 정적 방법과 LACP (Link Aggregation Control Protocol)라는 프로토콜을 사용하여으로 시작하는 역동적 인 방법이 있습니다.

역동적 인 방법을 채용 한 경우 각 네트워크 장비는 대량 인터페이스 동 선비에서 LACP 데이터 유닛을 정기적으로 교환하여 소통 불가능하지 것을 서로 확인하고 계속합니다. LACP 데이터 유닛 교환 단절 고장이 발생한 것으로 간주하고 해당 네트워크 장비는 사용 불가능 패킷 전송 수신은 나머지 인터페이스에 의해서만 이루어지게됩니다. 이 방법은 네트워크 장비간에 미디어 컨버터 등의 중계 장치가 존재하는 경우에도 중계 장치의 반대편 링크 다운을 감지 할 수 있다는 장점이 있습니다. 이 장에서는 LACP를 이용한 동적 링크 통합 기능을 처리 또는 입니다.

5.2 Ryu 응용 프로그램의 실행

소스의 설명은 차후에 하고, 우선 Ryu의 링크 애그리 게이션 응용 프로그램을 실행 해 봅니다.

Ryu 소스 트리에 포함되어 있는 simple_switch_lacp.py는 OpenFlow 1.0 전용 응용 프로그램이기 때문에 여기에서는 새롭게 OpenFlow 1.3에 대응 한 simple_switch_lacp_13.py을 만듭니다. 이 프로그램은 「[스위칭 허브](#)」 스위칭 허브 링크 어그리게이션 기능을 추가 한 응용 프로그램입니다.

소스 이름: simple_switch_lacp_13.py

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import lacplib
from ryu.lib.dpid import str_to_dpid
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```



```
class SimpleSwitchLacp13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'lacplib': lacplib.LacpLib}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitchLacp13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self._lacp = kwargs['lacplib']
        self._lacp.add(
            dpid=str_to_dpid('0000000000000001'), ports=[1, 2])

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                         ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPIInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                              actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                               match=match, instructions=inst)
        datapath.send_msg(mod)

    def del_flow(self, datapath, match):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        mod = parser.OFPFlowMod(datapath=datapath,
                               command=ofproto.OFPFC_DELETE,
```

```

        out_port=ofproto.OFPP_ANY,
        out_group=ofproto.OFPG_ANY,
        match=match)
datapath.send_msg(mod)

@set_ev_cls(lacplib.EventPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ether.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                               in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)

@set_ev_cls(lacplib.EventSlaveStateChanged, MAIN_DISPATCHER)
def _slave_state_changed_handler(self, ev):
    datapath = ev.datapath
    dpid = datapath.id
    port_no = ev.port
    enabled = ev.enabled
    self.logger.info("slave state changed port: %d enabled: %s",
                     port_no, enabled)
    if dpid in self.mac_to_port:
        for mac in self.mac_to_port[dpid]:
            match = datapath.ofproto_parser.OFPMatch(eth_dst=mac)
            self.del_flow(datapath, match)
        del self.mac_to_port[dpid]
    self.mac_to_port.setdefault(dpid, {})

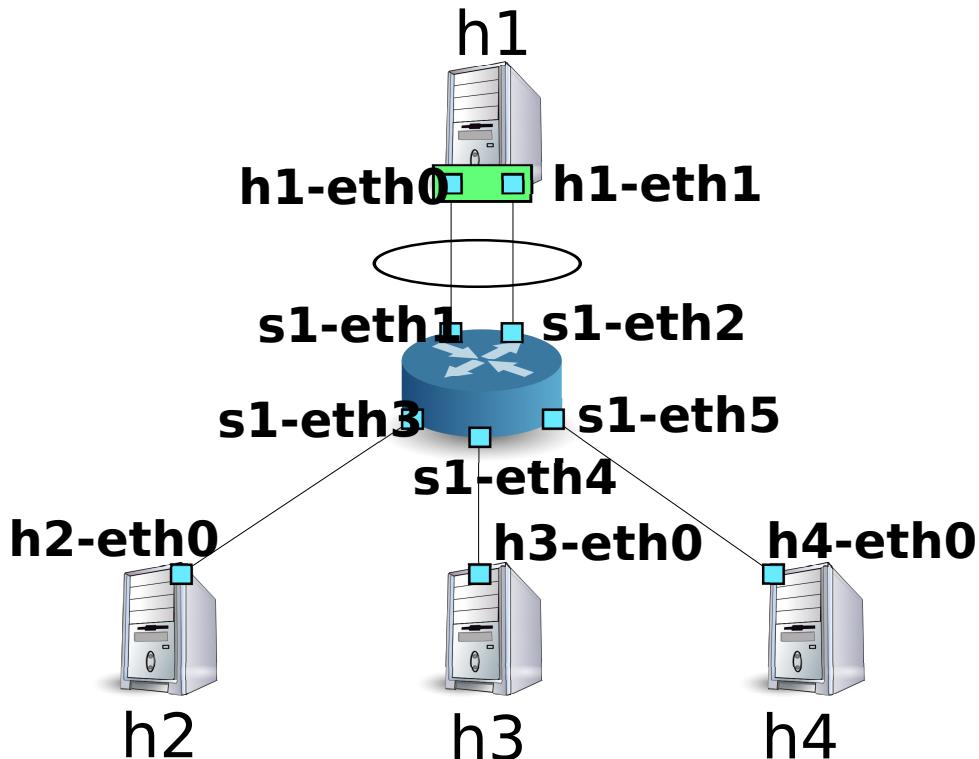
```

5.2.1 실험 환경 구축

OpenFlow 스위치 및 Linux 호스트 사이에서 링크 통합을 구성하여 봅시다.

VM 이미지 사용을 위한 환경 설정 및 로그인 방법 등은 「[스위칭 허브](#)」을 참조하십시오.

먼저 Mininet를 이용하여 아래 그림과 같은 토플로지를 만듭니다.



Mininet API를 호출하는 스크립트를 작성하고 필요한 토플로지를 구축 합니다.

소스 이름: link_aggregation.py

```

#!/usr/bin/env python

from mininet.cli import CLI
from mininet.link import Link
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.term import makeTerm

if '__main__' == __name__:
    net = Mininet(controller=RemoteController)

    c0 = net.addController('c0')

    s1 = net.addSwitch('s1')

    h1 = net.addHost('h1')
    h2 = net.addHost('h2', mac='00:00:00:00:00:22')
    h3 = net.addHost('h3', mac='00:00:00:00:00:23')
    h4 = net.addHost('h4', mac='00:00:00:00:00:24')

    Link(s1, h1)
    Link(s1, h1)
    Link(s1, h2)
    Link(s1, h3)
    Link(s1, h4)
  
```

```

net.build()
c0.start()
s1.start([c0])

net.terms.append(makeTerm(c0))
net.terms.append(makeTerm(s1))
net.terms.append(makeTerm(h1))
net.terms.append(makeTerm(h2))
net.terms.append(makeTerm(h3))
net.terms.append(makeTerm(h4))

CLI(net)

net.stop()

```

이 스크립트를 실행하여 호스트 h1과 스위치 s1 사이에 2 개의 링크가 존재하는 토플로지가 됩니다. net 명령으로 생성된 토플로지를 확인해야 할 수 있습니다.

```

ryu@ryu-vm:~$ sudo ./link_aggregation.py
Unable to contact the remote controller at 127.0.0.1:6633
mininet> net
c0
s1 lo:  s1-eth1:h1-eth0  s1-eth2:h1-eth1  s1-eth3:h2-eth0  s1-eth4:h3-eth0  s1-eth5:h4-eth0
h1 h1-eth0:s1-eth1  h1-eth1:s1-eth2
h2 h2-eth0:s1-eth3
h3 h3-eth0:s1-eth4
h4 h4-eth0:s1-eth5
mininet>

```

5.2.2 호스트 h1에서 링크 통합 구성

호스트 h1의 Linux에 필요한 사전 설정을 실시합니다. 절에서 명령 입력 호스트 h1의 xterm에서 실시해주세요.

먼저 링크 통합을 위한 드라이버 모듈을 로드합니다. Linux에서는 링크 통합 기능을 결합 드라이버가 담당하고 있습니다. 미리 드라이버의 설정 파일을 /etc/modprobe.d/bonding.conf로 작성해 수 있습니다.

파일 이름: /etc/modprobe.d/bonding.conf

```

alias bond0 bonding
options bonding mode=4

```

Node: h1:

```
root@ryu-vm:~# modprobe bonding
```

mode = 4는 LACP를 이용한 동적 링크 통합 할 것을 나타냅니다. 기본값이기 때문에 여기에서 설정을 선택하고 있습니다만, LACP 데이터 유닛 교환주기는 SLOW(30 초 간격) 배분 논리는 목적지 MAC 주소를 기반으로 할거야 게 설정되어 있습니다.

이어 bond0이라는 논리 인터페이스를 새로 만듭니다. 또한 bond0 MAC 주소로 적당한 값을 설정합니다.

Node: h1:

```
root@ryu-vm:~# ip link add bond0 type bond
root@ryu-vm:~# ip link set bond0 address 02:01:02:03:04:08
```

만든 논리 인터페이스의 그룹에 h1-eth0와 h1-eth1의 물리적 인터페이스를 참여시킵니다. 이 때, 물리적 인터페이스를 다운시켜 둘 필요가 있습니다. 또한 무작위로 결정되는 물리적 인터페이스의 MAC 주소를 알기 쉬운 값으로 갱신해야 합니다.

Node: h1:

```
root@ryu-vm:~# ip link set h1-eth0 down
root@ryu-vm:~# ip link set h1-eth0 address 00:00:00:00:00:11
root@ryu-vm:~# ip link set h1-eth0 master bond0
root@ryu-vm:~# ip link set h1-eth1 down
root@ryu-vm:~# ip link set h1-eth1 address 00:00:00:00:00:12
root@ryu-vm:~# ip link set h1-eth1 master bond0
```

논리 인터페이스에 IP 주소를 할당합니다. 여기에 10.0.0.1를 할당합니다. 또한 h1-eth0에 IP 주소 이 할당되어 있으므로 이를 제거합니다.

Node: h1:

```
root@ryu-vm:~# ip addr add 10.0.0.1/8 dev bond0
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1-eth0
```

마지막으로, 논리적 인터페이스를 향상시킵니다.

Node: h1:

```
root@ryu-vm:~# ip link set bond0 up
```

여기에서 각 인터페이스의 상태를 확인해야합니다.

Node: h1:

```
root@ryu-vm:~# ifconfig
bond0      Link encap:Ethernet HWaddr 02:01:02:03:04:08
           inet addr:10.0.0.1 Bcast:0.0.0.0 Mask:255.0.0.0
             UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:0 (0.0 B) TX bytes:1240 (1.2 KB)

h1-eth0    Link encap:Ethernet HWaddr 02:01:02:03:04:08
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:620 (620.0 B)

h1-eth1    Link encap:Ethernet HWaddr 02:01:02:03:04:08
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:620 (620.0 B)

lo         Link encap:Local Loopback
           inet addr:127.0.0.1 Mask:255.0.0.0
             UP LOOPBACK RUNNING MTU:16436 Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

논리적 인터페이스 bond0가 MASTER 물리 인터페이스 h1-eth0와 h1-eth1이 SLAVE되어 있는 것을 알 수 있습니다. 또한 bond0, h1-eth0, h1-eth1의 MAC 주소 스가 모두 동일하게되어 있는 것을 알 수 있습니다.

본딩 드라이버의 상태도 확인해야합니다.

Node: h1:

```
root@ryu-vm:~# cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.7.1 (April 27, 2011)
```

```

Bonding Mode: IEEE 802.3ad Dynamic link aggregation
Transmit Hash Policy: layer2 (0)
MII Status: up
MII Polling Interval (ms): 100
Up Delay (ms): 0
Down Delay (ms): 0

802.3ad info
LACP rate: slow
Min links: 0
Aggregator selection policy (ad_select): stable
Active Aggregator Info:
    Aggregator ID: 1
    Number of ports: 1
    Actor Key: 33
    Partner Key: 1
    Partner Mac Address: 00:00:00:00:00:00

Slave Interface: h1-eth0
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 00:00:00:00:00:11
Aggregator ID: 1
Slave queue ID: 0

Slave Interface: h1-eth1
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 00:00:00:00:00:12
Aggregator ID: 2
Slave queue ID: 0

```

LACP 데이터 유닛의 교환주기 (LACP rate : slow)와 배분 로직 설정 (Transmit Hash Policy : layer2 (0))를 확인할 수 있습니다. 또한 물리적 인터페이스 h1-eth0와 h1-eth1의 MAC 주소를 확인할 수 있습니다.
이상으로 호스트 h1의 사전 설정이 완료됩니다.

5.2.3 OpenFlow 버전 설정

스위치 s1의 OpenFlow의 버전을 1.3으로 설정합니다. 이 명령 입력 스위치 s1의 xterm에서 실시 하십시오.

Node: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

5.2.4 스위칭 허브의 실행

준비가 그래서 처음에 만든 Ryu 응용 프로그램을 실행합니다.

윈도우 제목이 「Node: c0 (root)」이다 xterm에서 다음 명령 을 실행합니다.

Node: c0:

```
ryu@ryu-vm:~$ ryu-manager ./simple_switch_lacp_13.py
loading app ./simple_switch_lacp_13.py
loading app ryu.controller.ofp_handler
creating context lacplib
instantiating app ./simple_switch_lacp_13.py
```

```
instantiating app ryu.controller.ofp_handler
...
```

호스트 h1은 30초에 한 번 LACP 데이터 단위를 전송합니다. 시작에서 시바 낙서하면 스위치는 호스트 h1에서 LACP 데이터 단위를 수신하여 작동 로그에 출력합니다.

Node: c0:

```
...
[LACP] [INFO] SW=0000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=1 the slave i/f has just been up.
[LACP] [INFO] SW=0000000000000001 PORT=1 the timeout time has changed.
[LACP] [INFO] SW=0000000000000001 PORT=1 LACP sent.
slave state changed port: 1 enabled: True
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=2 the slave i/f has just been up.
[LACP] [INFO] SW=0000000000000001 PORT=2 the timeout time has changed.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP sent.
slave state changed port: 2 enabled: True
...
```

로그는 다음을 나타냅니다.

- LACP received.

LACP 데이터 단위를 수신했습니다.

- the slave i/f has just been up.

비활성화 상태였던 포트가 활성화 상태로 변경되었습니다.

- the timeout time has changed.

LACP 데이터 단위 무선 통신 감시 시간이 변경되었습니다(이번 경우 초기 상태 0초에서 LONG_TIMEOUT_TIME 90초로 변경됩니다).

- LACP sent.

응답에 대한 LACP 데이터 단위를 전송했습니다.

- slave state changed ...

LACP 도서관에서 “EventSlaveStateChanged” 이벤트를 응용 프로그램이 수신되었습니다(이벤트의 자세한 내용은 아래 참조).

스위치는 호스트 h1에서 LACP 데이터 유닛을 수신 할 때마다 응답 용 LACP 데이터 단위 전송 신합니다.

Node: c0:

```
...
[LACP] [INFO] SW=0000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=1 LACP sent.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP sent.
...
```

흐름 항목을 확인하여 봅시다.

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=14.565s, table=0, n_packets=1, n_bytes=124, idle_timeout=90, send_flow_rem
cookie=0x0, duration=14.562s, table=0, n_packets=1, n_bytes=124, idle_timeout=90, send_flow_rem
cookie=0x0, duration=24.821s, table=0, n_packets=2, n_bytes=248, priority=0 actions=CONTROLLER:64
```

스위치는

- h1의 h1-eth1(입력 포트가 s1-eth2에서 MAC 주소가 00:00:00:00:00:12)에서 LACP 데이터 유닛 (ether-type이 0x8809)가 보내져 오면 Packet-In 메시지를 보냄

- h1의 h1-eth0(입력 포트가 s1-eth1에서 MAC 주소가 00:00:00:00:00:11)에서 LACP 데이터 유닛(ether-type가 0x8809)가 보내져 오면 Packet-In 메시지를 보냄
- 「스위칭 허브」와 같은 Table-miss 항목

세 가지 항목이 등록되어 있습니다.

5.2.5 링크 통합 기능 확인

통신 속도 향상

우선 링크 통합에 의한 통신 속도의 향상을 확인합니다. 통신에 따라 여러 링크를 구사하는 모습을 보고하자.

먼저 호스트 h2에서 호스트 h1 대해 ping을 실행합니다.

Node: h2:

```
root@ryu-vm:~# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_req=1 ttl=64 time=93.0 ms
64 bytes from 10.0.0.1: icmp_req=2 ttl=64 time=0.266 ms
64 bytes from 10.0.0.1: icmp_req=3 ttl=64 time=0.075 ms
64 bytes from 10.0.0.1: icmp_req=4 ttl=64 time=0.065 ms
...
```

ping을 계속 채 스위치 s1의 항목을 확인합니다.

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=22.05s, table=0, n_packets=1, n_bytes=124, idle_timeout=90, send_flow_rem p
cookie=0x0, duration=22.046s, table=0, n_packets=1, n_bytes=124, idle_timeout=90, send_flow_rem p
cookie=0x0, duration=33.046s, table=0, n_packets=6, n_bytes=472, priority=0 actions=CONTROLLER:66
cookie=0x0, duration=3.259s, table=0, n_packets=3, n_bytes=294, priority=1,in_port=3,dl_dst=02:00:00:00:00:03
cookie=0x0, duration=3.262s, table=0, n_packets=4, n_bytes=392, priority=1,in_port=1,dl_dst=00:0c:29:ff:ff:01
```

방금 확인한 시점에서 두 항목이 추가되어 있습니다. duration 값이 작은 4 번째와 5 번째 항목입니다.

각각

- 3 번 포트(s1-eth3, 즉 h2 대향 인터페이스)에서 h1의 bond0에게 파 패킷을 수신하면 1 번 포트(s1-eth1)에서 출력
- 1 번 포트(s1-eth1)에서 h2에게 패킷을 받으면 3 번 포트(s1-eth3)에서 출력하기

라는 항목입니다. h2와 h1 사이의 통신에는 s1-eth1이 사용된 것을 알 수 있습니다.

그런 다음 호스트 h3에서 호스트 h1 대해 ping을 실행합니다.

Node: h3:

```
root@ryu-vm:~# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_req=1 ttl=64 time=91.2 ms
64 bytes from 10.0.0.1: icmp_req=2 ttl=64 time=0.256 ms
64 bytes from 10.0.0.1: icmp_req=3 ttl=64 time=0.057 ms
64 bytes from 10.0.0.1: icmp_req=4 ttl=64 time=0.073 ms
...
```

ping을 계속 채 스위치 s1의 항목을 확인합니다.

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=99.765s, table=0, n_packets=4, n_bytes=496, idle_timeout=90, send_flow_rem ...
cookie=0x0, duration=99.761s, table=0, n_packets=4, n_bytes=496, idle_timeout=90, send_flow_rem ...
cookie=0x0, duration=110.761s, table=0, n_packets=10, n_bytes=696, priority=0 actions=CONTROLLER...
cookie=0x0, duration=80.974s, table=0, n_packets=82, n_bytes=7924, priority=1,in_port=3,d1_dst=0...
cookie=0x0, duration=2.677s, table=0, n_packets=2, n_bytes=196, priority=1,in_port=2,d1_dst=00:0...
cookie=0x0, duration=2.675s, table=0, n_packets=1, n_bytes=98, priority=1,in_port=4,d1_dst=02:0...
cookie=0x0, duration=80.977s, table=0, n_packets=83, n_bytes=8022, priority=1,in_port=1,d1_dst=0...
```

방금 확인한 시점에서 두 흐름 항목이 추가되어 있습니다. duration 값이 작은 다섯 번째와 여섯 번째 항목입니다.

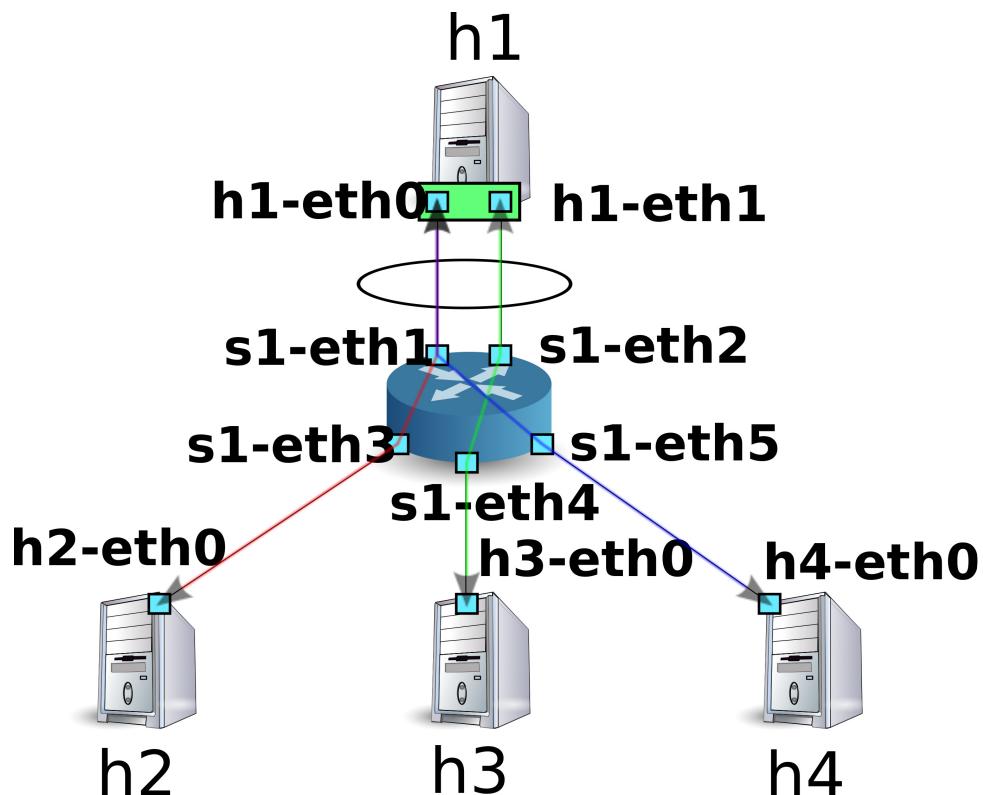
각각

- 2 번 포트 (s1-eth2)에서 h3에게 패킷을 수신하면 4 번 포트 (s1-eth4)에서 출력하기
- 4 번 포트 (s1-eth4 즉 h3의 대향 인터페이스)에서 h1의 bond0에게의 파 패킷을 수신하면 2 번 포트 (s1-eth2)에서 출력

라는 흐름 항목입니다. h3와 h1 사이의 통신에는 s1-eth2가 사용 된 것을 알 수 있습니다.

물론 호스트 h4에서 호스트 h1로도 ping을 수행 할 수 있습니다. 지금까지와 마찬가지로 새로운 흐름 항목이 등록되어 h4와 h1 사이의 통신에는 s1-eth1이 사용됩니다.

대상 호스트	사용 포트
h2	1
h3	2
h4	1



이상과 같이 통신에 따라 여러 링크를 구사하는 모습을 확인할 수 있었습니다.

결합 허용 향상

다음 링크 통합에 의한 감수성 향상을 확인합니다. 현재 상황은 h2와 h4가 h1과 통신 할 때 s1-eth2를 h3이 h1과 통신 할 때 s1-eth1을 사용하여 있습니다.

여기서 s1-eth1의 대량 인터페이스이다 h1-eth0를 링크 통합 그룹에서 이탈시킵니다.

Node: h1:

```
root@ryu-vm:~# ip link set h1-eth0 nomaster
```

h1-eth0가 중지함으로써 호스트 h3에서 호스트 h1에 ping이 소통 불가능 있습니다. 무 통신 감시 시간 90초가 경과하면 컨트롤러의 동작 로그에 다음과 같은 메시지가 출력됩니다.

Node: c0:

```
...
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP sent.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP sent.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP sent.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP sent.
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP exchange timeout has occurred.
slave state changed port: 1 enabled: False
...
```

「LACP exchange timeout has occurred.」무 통신 감시 시간에 도달 것을 나타냅니다. 여기에서는 학습 한 MAC 주소 전송의 흐름 항목을 삭제하는 것으로, 스위치를 시작한 직후의 상태로 되돌립니다.

새로운 통신이 발생하면 새로운 MAC 주소를 학습하고 살아있는 링크만을 이용한 흐름 항목이 다시 등록 됩니다.

호스트 h3와 호스트 h1 사이도 새로운 흐름 항목이 등록되어

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=364.265s, table=0, n_packets=13, n_bytes=1612, idle_timeout=90, send_flow_replies=1
cookie=0x0, duration=374.521s, table=0, n_packets=25, n_bytes=1830, priority=0 actions=CONTROLLER:0
cookie=0x0, duration=5.738s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=3,d1_dst=02:00:00:00:00:01
cookie=0x0, duration=6.279s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=2,d1_dst=00:00:00:00:00:02
cookie=0x0, duration=6.281s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=5,d1_dst=02:00:00:00:00:01
cookie=0x0, duration=5.506s, table=0, n_packets=5, n_bytes=434, priority=1,in_port=4,d1_dst=02:00:00:00:00:01
cookie=0x0, duration=5.736s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=2,d1_dst=00:00:00:00:00:02
cookie=0x0, duration=6.504s, table=0, n_packets=6, n_bytes=532, priority=1,in_port=2,d1_dst=00:00:00:00:00:01
```

호스트 h3 중지했다 ping이 다시 시작합니다.

Node: h3:

```
...
64 bytes from 10.0.0.1: icmp_req=144 ttl=64 time=0.193 ms
64 bytes from 10.0.0.1: icmp_req=145 ttl=64 time=0.081 ms
64 bytes from 10.0.0.1: icmp_req=146 ttl=64 time=0.095 ms
64 bytes from 10.0.0.1: icmp_req=237 ttl=64 time=44.1 ms
64 bytes from 10.0.0.1: icmp_req=238 ttl=64 time=2.52 ms
64 bytes from 10.0.0.1: icmp_req=239 ttl=64 time=0.371 ms
64 bytes from 10.0.0.1: icmp_req=240 ttl=64 time=0.103 ms
64 bytes from 10.0.0.1: icmp_req=241 ttl=64 time=0.067 ms
...
```

이상과 같이 일부 링크에 고장이 발생한 경우에도 다른 링크를 사용하여 자동으로 복구 할 수 있는지 확인 할 수 있었습니다.

5.3 Ryu의 링크 애그리 게이션 기능의 구현

OpenFlow를 이용한 있어 등 같이 링크 통합 기능을 제공하는지 알아 보겠습니다.

LACP를 이용한 링크 어그리 게이션은 「LACP 데이터 유닛의 교환이 성공적으로 이루어지고 있는 동안은 해당 물리적 인터페이스는 사용」「LACP 데이터 유닛의 교환이 두절되면 해당 물리적 인터페이스는 무효」라고 거절 무용입니다. 물리적 인터페이스를 비활성화하는 것은, 그 인터페이스를 사용 용하는 흐름 항목이 존재하지 않는다는 것도 있습니다. 따라서,

- LACP 데이터 단위를 수신하면 응답을 작성하여 보냄
- LACP 데이터 단위가 일정 시간 수신 할 수 없는 경우 해당 물리적 인터페이스를 사용 흐름 항목을 삭제하고 이후 그 인터페이스를 사용하는 흐름 항목을 등록
- 무효가 된 물리적 인터페이스에서 LACP 데이터 유닛을 받은 경우 당해 인터페이스를 다시 활성화
- LACP 데이터 단위 이외의 패킷은 「스위칭 허브」처럼 학습 · 전송

하는 처리를 구현하면 링크 통합의 기본 동작이 가능해 또는 입니다. LACP에 관련되는 부분과 그렇지 않은 부분이 명확하게 나뉘어 있기 때문에 LACP에 관한 부분을 LACP 라이브러리로 잘라, 그렇지 않은 부분은 「스위칭 허브」스위칭 허브를 확장하는 형태로 구현합니다 입니다.

LACP 데이터 유닛 수신시 응답 작성 · 전송 흐름 항목만으로는 실현 불가능 이기 때문에 Packet-In 메시지를 사용하여 OpenFlow 컨트롤러에서 처리를 수행 있습니다.

주석: LACP 데이터 유닛을 교체하려는 물리 인터페이스는 그 역할에 따라 ACTIVE와 PASSIVE로 분류됩니다. ACTIVE는 일정 시간마다 LACP 데이터 유닛을 보내고 소통을 능동적으로 확인합니다. PASSIVE는 ACTIVE에서 전송된 LACP 데이터 단위를 수신했을 때 응답을 반환하여 소통을 수동적으로 확인합니다.

Ryu 링크 어그리 게이션 애플리케이션은 PASSIVE 모드 만 실 이렇게하고 있습니다.

일정 시간 LACP 데이터 단위를 받지 못한 경우 해당 물리적 인터페이스를 해제한다는 처리는 LACP 데이터 단위 Packet-In시키는 흐름 항목에 idle_timeout을 설정하고 만료시 FlowRemoved 메시지를 보내도록 할 하여 OpenFlow 컨트롤러에서 해당 인터페이스를 사용할 때의 대처를 할 수 있습니다.

비활성화 된 인터페이스에서 LACP 데이터 유닛의 교환이 재개 된 경우의 처리는 LACP 데이터 유닛 수신시의 Packet-In 메시지 처리기에서 해당 인터페이스의 활성화 / 비활성화 상태를 확인 · 수정하여 제공합니다.

물리적 인터페이스가 무효가 되었을 때, OpenFlow 컨트롤러의 처리로 “해당 인터페이스를 사용하는 흐름 항목 삭제”뿐만 좋을 듯 보네 합니다만, 그러면 충분하지 않습니다.

예를 들어 3 개의 물리적 인터페이스를 그룹화하여 사용하는 논리적 인터페이스가 있고 배분 논리가 “효과적인 인터페이스 수에 의한 MAC 주소 스 나머지 “라고되어 있는 경우를 가정합니다.

인터페이스1	인터페이스2	인터페이스3
MAC 주소의 나머지:0	MAC 주소의 나머지:1	MAC 주소의 나머지:2

그리고 각 물리적 인터페이스를 사용하는 흐름 항목이 다음과 같이 3 개씩 등록되어 있었다고 합니다.

인터페이스1	인터페이스2	인터페이스3
주소:00:00:00:00:00:00	주소:00:00:00:00:00:01	주소:00:00:00:00:00:02
주소:00:00:00:00:00:03	주소:00:00:00:00:00:04	주소:00:00:00:00:00:05
주소:00:00:00:00:00:06	주소:00:00:00:00:00:07	주소:00:00:00:00:00:08

여기서 인터페이스 1가 비활성화 된 경우 「효과적인 인터페이스 수에 의한 MAC 주소의 나머지」라는 배분 논리에 따라 다음과 같이 분류 수없는 하여야합니다.

인터페이스1	인터페이스2	인터페이스3
비활성화	MAC 주소의 나머지:0	MAC 주소의 나머지:1

인터페이스1	인터페이스2	인터페이스3
주소:00:00:00:00:00:00	주소:00:00:00:00:00:01	
주소:00:00:00:00:00:02	주소:00:00:00:00:00:03	
주소:00:00:00:00:00:04	주소:00:00:00:00:00:05	
주소:00:00:00:00:00:06	주소:00:00:00:00:00:07	
주소:00:00:00:00:00:08		

인터페이스 1을 사용하고 있었다 흐름 항목뿐만 아니라 인터페이스 2 또는 인터페이스 3의 흐름 항목도 다시 작성해야 볼 수 있습니다. 이것은 물리적 인터페이스를 사용할 때뿐만 아니라 활성화되었을 때도 마찬가지입니다.

따라서, 물리적 인터페이스의 활성화 / 비활성화 상태가 변경되었을 경우의 처리는 당해 물리적 인터페이스가 속한 논리 인터페이스에 포함 된 모든 물리적 인 페이스를 사용하는 흐름 항목을 삭제한다고 합니다.

주석: 배분 논리에 대해서는 사양으로 정해져 있지 않고, 각 기기의 구현에 맡길 수 있습니다. Ryu 링크 어그리게이션 애플리케이션은 자신의 모습 분할 처리를 하지 않고 대향 장치에 의해 배분 된 경로를 사용하고 있습니다입니다.

여기에는 다음과 같은 기능을 구현합니다.

LACP 라이브러리

- LACP 데이터 단위를 수신하면 응답을 작성하여 보냄
- LACP 데이터 유닛의 수신이 두 절되면 해당 물리적 인터페이스를 무효로 간주, 스위칭 허브에 통지 한다
- LACP 데이터 유닛의 수신이 재개되면 대응하는 물리 인터페이스를 유효한 것으로 간주 스위칭 허브에 통지한다

스위칭 허브

- LACP 라이브러리의 통지를 받아 초기화가 필요한 흐름 항목을 삭제
- LACP 데이터 단위 이외의 패킷은 종래대로 학습 · 전송

LACP 라이브러리 및 스위칭 허브 소스 코드는 Ryu 소스 트리에 있습니다.

ryu/lib/lacplib.py

ryu/app/simple_switch_lacp.py

주석: simple_switch_lacp.py는 OpenFlow 1.0 전용 응용 프로그램이기 때문에 이 장에서는 「Ryu 응용 프로그램의 실행」으로 보여 주었다 OpenFlow 1.3에 대응 한 simple_switch_lacp_13.py 기반으로 응용 프로그램 자세한 내용을 설명합니다.

5.3.1 LACP 라이브러리 구현

다음 절에서는 위의 기능이 LACP 라이브러리에서 어떻게 구현되어 누가보고갑니다. 또한 인용 된 소스는 발췌입니다. 전체 그림에 대해서는 실 때 소스를 참조하십시오.

논리적 인터페이스 작성

링크 통합 기능을 사용하려면 어떤 네트워크 기기에서 어떤 인터페이스를 어떤 그룹으로 묶을 것인가하는 설정을 일 전에 가서해야합니다. LACP 라이브러리는 다음과 같은 방법으로 이 설정을 합니다.

```
def add(self, dpid, ports):
    #
    assert isinstance(ports, list)
    assert 2 <= len(ports)
    ifs = {}
    for port in ports:
        ifs[port] = {'enabled': False, 'timeout': 0}
    bond = {}
```

```
bond[dpid] = ifs
self._bonds.append(bond)
```

인수의 내용은 다음과 같습니다.

dpid

OpenFlow 스위치의 데이터 경로 ID를 지정합니다.

ports

그룹화 할 포트 번호 목록을 지정합니다.

이 메서드를 호출하여 LACP 라이브러리는 지정된 데이터 경로 ID의 OpenFlow 스위치의 지정된 포트를 하나의 그룹으로 간주합니다. 여러 그룹을 만들려면 반복 add() 메서드를 호출 있습니다. 또한 논리적 인터페이스에 할당 된 MAC 주소는 OpenFlow 스위치가 가지는 LOCAL 포트와 동일하게 자동으로 사용됩니다.

참고: OpenFlow 스위치에 스위치 자신의 기능으로 링크 통합 기능을 제공하는 것도 있습니다 (Open vSwitch 등). 여기에서는 그러한 스위치 자체의 기능은 사용하지 않고, OpenFlow 컨트롤러의 제어에 의해 링크 통합 기능을 제공합니다.

Packet-In 처리

「스위칭 허브」은 대상의 MAC 주소가 미 학 학습의 경우 수신 된 패킷을 쇄도합니다. LACP 데이터 유닛은 인접해야 하는 네트워크 장비간에만 교환되어야 하며, 다른 기기에 전송해 버리면 리 잉크 어 그리 게이션 기능이 제대로 작동하지 않습니다. 그래서 “Packet-In 수신 패킷이 LACP 데이터 유닛이라면 차단하고 LACP 데이터 유닛 이외의 패킷이면 스위칭 허브의 동작에 맡긴다 ‘라는 처리를 실시해, 스위칭 허브는 LACP 데이터 단위를 보이지 않게합니다.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, evt):
    """PacketIn event handler. when the received packet was LACP,
    proceed it. otherwise, send a event."""
    req_pkt = packet.Packet(evt.msg.data)
    if slow.lacp in req_pkt:
        (req_lacp, ) = req_pkt.get_protocols(slow.lacp)
        (req_eth, ) = req_pkt.get_protocols(ethernet.ethernet)
        self._do_lacp(req_lacp, req_eth.src, evt.msg)
    else:
        self.send_event_to_observers(EventPacketIn(evt.msg))
```

이벤트 처리기 자체는 「스위칭 허브」이라고 같습니다. 받은 메시지에 LACP 데이터 유닛이 포함되어 있는지 여부에 처리를 분기시키고 있습니다.

LACP 데이터 단위가 포함 된 경우 LACP 라이브러리 LACP 데이터 단위 접수 신호 처리합니다. LACP 데이터 단위가 포함되지 않은 경우 send_event_to_observers()라는 메서드를 부르고 있습니다. 이것은 ryu.base.app_manager.RyuApp 클래스에 정의 된 이벤트를 전송하기 위해 방법입니다.

「스위칭 허브」는 Ryu에 정의 된 OpenFlow 메시지 수신 그 벤트에 대해 언급했지만, 사용자가 직접 이벤트를 정의 할 수 있습니다. 위 소스에서 보내는 EventPacketIn라는 이벤트는 LACP 라이브러리 리에서 만든 사용자 정의 이벤트입니다.

```
class EventPacketIn(event.EventBase):
    """a PacketIn event class using except LACP."""
    def __init__(self, msg):
        """initialization."""
        super(EventPacketIn, self).__init__()
        self.msg = msg
```

사용자 정의 이벤트는 ryu.controller.event.EventBase 클래스를 상속하여 만든 합니다. 이벤트 클래스에 내포하는 데이터에 제한은 없습니다. EventPacketIn 클래스는 Packet-In 메시지 받은 ryu.ofproto.OFPPacketIn 인스턴스 스를 그대로 사용하고 있습니다.

사용자 정의 이벤트를 수신하는 방법에 대해서는 후술합니다.

포트를 활성화 / 비활성화 상태 변경에 따른 처리

LACP 라이브러리 LACP 데이터 유닛 수신 처리는 다음 작업으로 구성되어 있습니다.

1. LACP 데이터 단위를 받은 포트가 비활성화 상태라면 활성화 상태로 변경하고 상태가 변경되었음을 이벤트 통지합니다.
2. 비활성 시간 초과 대기 시간이 변경된 경우 LACP 데이터 단위 받을 때 Packet-In을 보낼 흐름 항목을 다시 등록합니다.
3. 받은 LACP 데이터 단위에 대한 응답을 작성하고 보냅니다.

2. 처리 내용은 아래의 「LACP 데이터 단위 Packet-In시키는 흐름 항목의 등록」 그리고, 3.의 처리 내용은 아래의 「LACP 데이터 단위의 송수신 처리」로 각각 설명합니다. 여기에서는 1.의 처리에 대해 설명합니다.

```
def _do_lacp(self, req_lacp, src, msg):
    # ...

    # when LACP arrived at disabled port, update the status of
    # the slave i/f to enabled, and send a event.
    if not self._get_slave_enabled(dpid, port):
        self.logger.info(
            "SW=%s PORT=%d the slave i/f has just been up.",
            dpid_to_str(dpid), port)
        self._set_slave_enabled(dpid, port, True)
        self.send_event_to_observers(
            EventSlaveStateChanged(datapath, port, True))
```

`_get_slave_enabled()` 메서드는 지정된 스위치의 지정된 포트가 유효한지 여부 하나를 가져옵니다. `_set_slave_enabled()` 메서드는 지정된 스위치의 지정된 포트를 활성화 / 비활성화 상태를 설정합니다.

위의 소스에서 비활성 상태의 포트에서 LACP 데이터 유닛을 받은 경우 포트 상태가 변경되었다는 것을 나타내는 `EventSlaveStateChanged`라는 사용자 정의 이벤트를 전송합니다.

```
class EventSlaveStateChanged(event.EventBase):
    """A event class that notifies the changes of the statuses of the
    slave i/fs."""
    def __init__(self, datapath, port, enabled):
        """Initialization."""
        super(EventSlaveStateChanged, self).__init__()
        self.datapath = datapath
        self.port = port
        self.enabled = enabled
```

`EventSlaveStateChanged` 이벤트는 포트가 활성화되었을 때 다른 포트 이 비활성화 된 경우에도 전송됩니다. 비활성화했을 때의 처리는 「FlowRemoved 메시지의 수신 처리」으로 구현되어 있습니다.

`EventSlaveStateChanged` 클래스에는 다음 정보가 포함됩니다.

- 포트를 활성화 / 비활성화 상태 변경이 발생한 OpenFlow 스위치
- 활성화 / 비활성화 상태 변경이 발생한 포트 번호
- 변경 후 상태

LACP 데이터 단위 Packet-In시키는 흐름 항목의 등록

LACP 데이터 유닛의 교환주기는 FAST(초당)와 SLOW(30 초마다)의 2 종류가 정의되어 있습니다. 링크 통합의 사양에서는 교환주기의 3 배의 시간 무 통 신호 상태가 계속 된 경우, 그 인터페이스는 링크 집계 그룹에서 제외 된 패킷의 전송에 사용되지 않습니다.

LACP 라이브러리는 LACP 데이터 단위 받을 때 Packet-In 시키는 흐름 항목 반면 교환주기의 3 배의 시간 (SHORT_TIMEOUT_TIME은 3 초, LONG_TIMEOUT_TIME은 90 초)을 idle_timeout로 설정하여 비활성 모니터링을 실시하고 있습니다.

교환주기가 변경된 경우 idle_timeout 시간도 다시 설정해야 하므로 LACP 라이브러리는 다음과 같은 구현을 하고 있습니다.

```
def _do_lacp(self, req_lacp, src, msg):
    # ...

    # set the idle_timeout time using the actor state of the
    # received packet.
    if req_lacp.LACP_STATE_SHORT_TIMEOUT == \
        req_lacp.actor_state_timeout:
        idle_timeout = req_lacp.SHORT_TIMEOUT_TIME
    else:
        idle_timeout = req_lacp.LONG_TIMEOUT_TIME

    # when the timeout time has changed, update the timeout time of
    # the slave i/f and re-enter a flow entry for the packet from
    # the slave i/f with idle_timeout.
    if idle_timeout != self._get_slave_timeout(dpid, port):
        self.logger.info(
            "SW=%s PORT=%d the timeout time has changed.",
            dpid_to_str(dpid), port)
        self._set_slave_timeout(dpid, port, idle_timeout)
        func = self._add_flow.get(ofproto.OFP_VERSION)
        assert func
        func(src, port, idle_timeout, datapath)

    # ...
```

_get_slave_timeout () 메서드는 지정된 스위치의 지정된 포트의 현 재의 idle_timeout 값을 가져옵니다. _set_slave_timeout () 메서드는 지정된 스위치의 지정된 포트에서 idle_timeout 값을 등록합니다. 초기 상태 및 링크 통합 그룹에서 제외 된 경우에는 idle_timeout 값은 0 설정되어 있기 때문에 새로운 LACP 데이터 유닛을 받은 경우 교환주기가 어느 심지어 흐름 항목을 등록합니다.

사용하는 OpenFlow 버전에 따라 OFPFFlowMod 클래스의 생성자 인수가 다르기 때문에 버전에 따라 유동 항목 등록 방법을 스크리닝입니다. 다음은 OpenFlow 1.2 이상에서 사용할 흐름 항목 등록 방법입니다.

```
def _add_flow_v1_2(self, src, port, timeout, datapath):
    """enter a flow entry for the packet from the slave i/f
    with idle_timeout. for OpenFlow ver1.2 and ver1.3."""
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    match = parser.OFPMatch(
        in_port=port, eth_src=src, eth_type=ether.ETH_TYPE_SLOW)
    actions = [parser.OFPActionOutput(
        ofproto.OFPP_CONTROLLER, ofproto.OFPCML_MAX)]
    inst = [parser.OFPIInstructionActions(
        ofproto.OFPIIT_APPLY_ACTIONS, actions)]
    mod = parser.OFPPFlowMod(
        datapath=datapath, command=ofproto.OFPFC_ADD,
        idle_timeout=timeout, priority=65535,
        flags=ofproto.OFPFF_SEND_FLOW_REM, match=match,
        instructions=inst)
    datapath.send_msg(mod)
```

위 소스에서 「대향 인터페이스에서 LACP 데이터 유닛을 받은 경우 Packet-In 한다」라고 하는 흐름 항목을 비활성 감시 시간 월 최고 우선 순위로 설정하고 있습니다.

LACP 데이터 단위의 송수신 처리

LACP 데이터 단위받을 때 「포트를 활성화 / 비활성화 상태 변경에 따른 처리」 또는 「LACP 데이터 단위 Packet-In시키는 흐름 항목의 등록」을 실시했다 후 응답에 대한 LACP 데이터 단위를 만들고 보냅니다.

```
def _do_lacp(self, req_lacp, src, msg):
    # ...

    # create a response packet.
    res_pkt = self._create_response(datapath, port, req_lacp)

    # packet-out the response packet.
    out_port = ofproto.OFPP_IN_PORT
    actions = [parser.OFPActionOutput(out_port)]
    out = datapath.ofproto_parser.OFPPacketOut(
        datapath=datapath, buffer_id=ofproto.OFP_NO_BUFFER,
        data=res_pkt.data, in_port=port, actions=actions)
    datapath.send_msg(out)
```

위 소스에서 호출되는 `_create_response()` 메서드는 응답용 패킷 만들기 처리입니다. 그 중에 호출되는 `_create_lacp()` 메서드 응답에 대한 LACP 데이터 유닛을 만들고 있습니다. 작성한 응답용 패킷은 LACP 데이터 단위 수신 포트에서 Packet-Out시킵니다.

LACP 데이터 단위에는 전송(Actor)의 정보와 수신자(Partner)의 정보를 설정합니다. 받은 LACP 데이터 단위 보내는 정보에는 대향 인터페이스 정보가 기재되어 있으므로, OpenFlow 스위치에서 응답을 돌려줄 때 그것을 받는 정보로 설정합니다.

```
def _create_lacp(self, datapath, port, req):
    """create a LACP packet."""
    actor_system = datapath.ports[datapath.ofproto.OFPP_LOCAL].hw_addr
    res = slow.lacp(
        # ...
        partner_system_priority=req.actor_system_priority,
        partner_system=req.actor_system,
        partner_key=req.actor_key,
        partner_port_priority=req.actor_port_priority,
        partner_port=req.actor_port,
        partner_state_activity=req.actor_state_activity,
        partner_state_timeout=req.actor_state_timeout,
        partner_state_aggregation=req.actor_state_aggregation,
        partner_state_synchronization=req.actor_state_synchronization,
        partner_state_collecting=req.actor_state_collecting,
        partner_state_distributing=req.actor_state_distributing,
        partner_state_defaulted=req.actor_state_defaulted,
        partner_state_expired=req.actor_state_expired,
        collector_max_delay=0)
    self.logger.info("SW=%s PORT=%d LACP sent.",
                     dpid_to_str(datapath.id), port)
    self.logger.debug(str(res))
    return res
```

FlowRemoved 메시지의 수신 처리

지정된 시간동안 LACP 데이터 유닛의 교환이 이루어 않으면 OpenFlow 스 계약은 FlowRemoved 메시지를 OpenFlow 컨트롤러에 보냅니다.

```
@set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
def flow_removed_handler(self, evt):
    """FlowRemoved event handler. when the removed flow entry was
    for LACP, set the status of the slave i/f to disabled, and
    send a event."""
    msg = evt.msg
```

```

datapath = msg.datapath
ofproto = datapath.ofproto
dpid = datapath.id
match = msg.match
if ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
    port = match.in_port
    dl_type = match.dl_type
else:
    port = match['in_port']
    dl_type = match['eth_type']
if ether.ETH_TYPE_SLOW != dl_type:
    return
self.logger.info(
    "SW=%s PORT=%d LACP exchange timeout has occurred.",
    dpid_to_str(dpid), port)
self._set_slave_enabled(dpid, port, False)
self._set_slave_timeout(dpid, port, 0)
self.send_event_to_observers(
    EventSlaveStateChanged(datapath, port, False))

```

FlowRemoved 메시지를 수신하면 OpenFlow 컨트롤러 _set_slave_enabled() 메서드를 사용하여 포트의 비활성 상태를 설정하고 _set_slave_timeout() 메서드를 사용하여 idle_timeout 값을 0으로 설정하고 send_event_to_observers() 메서드를 사용하여 EventSlaveStateChanged 이벤트를 보냅니다.

5.3.2 응용 프로그램 구현

「Ryu 응용 프로그램의 실행」에 나와있는 OpenFlow 1.3 대응의 링크 애그리 게슨 애플리케이션 (simple_switch_lacp_13.py)와 「스위칭 허브」스위칭 허브의 차이를 차례로 설명 합니다.

「_CONTEXTS」설정

ryu.base.app_manager.RyuApp을 계승 한 Ryu 응용 프로그램은 「_CONTEXTS」 사전에 다른 Ryu 응용 프로그램을 설정하여 다른 응용 프로그램을 별도의 스레드에서 실행시킬 수 있습니다. 여기에서는 LACP 라이브러리 LacpLib 클래스를 「lacplib」라는 이름으로 「_CONTEXTS」로 설정합니다.

```

from ryu.lib import lacplib

# ...

class SimpleSwitchLacp13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'lacplib': lacplib.LacpLib}

# ...

```

「_CONTEXTS」로 설정 한 응용 프로그램은 __init__() 메서드 kwargs에서 인스턴스를 얻을 수 있습니다.

```

# ...
def __init__(self, *args, **kwargs):
    super(SimpleSwitchLacp13, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self._lacp = kwargs['lacplib']
# ...

```

라이브러리의 기본

「_CONTEXTS」로 설정하여 LACP 라이브러리의 초기 구성은 수행합니다. 초기 설정은 LACP 라이브러리가 제공하는 add() 메소드를 실행합니다. 여기에 다음 값을 설정 있습니다.

매개변수	값	설명
dpid	str_to_dpid('0000000000000001')	데이터 경로 ID
ports	[1, 2]	그룹화하는 포트 목록

이 설정은 데이터 경로 ID 「0000000000000001」의 OpenFlow 스위치의 포트1과 포트2가 하나의 링크 통합 그룹으로 작동합니다.

```
# ...
    self._lacp = kwargs['lacplib']
    self._lacp.add(
        dpid=str_to_dpid('0000000000000001'), ports=[1, 2])
# ...
```

사용자 정의 이벤트를 수신하는 방법

LACP 라이브러리 구현에서 설명한대로 LACP 라이브러리는 LACP 데이터 유닛 트가 포함되지 않은 Packet-In 메시지를 EventPacketIn라는 사용자 정의 이벤트로 보냅니다. 사용자 정의 이벤트의 이벤트 처리기도 Ryu 제공합니다 이벤트 처리기처럼 ryu.controller.handler.set_ev_cls 데코레이터로 장식합니다.

```
@set_ev_cls(lacplib.EventPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    # ...
```

또한 LACP 라이브러리는 포트를 활성화 / 비활성화 상태가 변경되면 EventSlaveStateChanged 이벤트를 송신하기 때문에, 이쪽도 이벤트 핸드 라를 만들어둡니다.

```
@set_ev_cls(lacplib.EventSlaveStateChanged, lacplib.LAG_EV_DISPATCHER)
def _slave_state_changed_handler(self, ev):
    datapath = ev.datapath
    dpid = datapath.id
    port_no = ev.port
    enabled = ev.enabled
    self.logger.info("slave state changed port: %d enabled: %s",
                     port_no, enabled)
    if dpid in self.mac_to_port:
        for mac in self.mac_to_port[dpid]:
            match = datapath.ofproto_parser.OFPMatch(eth_dst=mac)
            self.del_flow(datapath, match)
        del self.mac_to_port[dpid]
    self.mac_to_port.setdefault(dpid, {})
```

이 절의 시작 부분에서 설명한대로 포트를 활성화 / 비활성화 상태가 변경됩니다 논리적 인터페이스를 통오버 패킷이 실제로 사용하는 물리적 인터페이스가 변경될 가능성이 있어입니다. 따라서 등록된 흐름 항목을 모두 삭제하고 있습니다.

```
def del_flow(self, datapath, match):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    mod = parser.OFPFlowMod(datapath=datapath,
                           command=ofproto.OFPFC_DELETE,
                           match=match)
    datapath.send_msg(mod)
```

흐름 항목의 삭제는 OFPFlowMod 클래스의 인스턴스로 합니다.

이상과 같이, 링크 통합 기능을 제공하는 라이브러리와 라이브러리 를 사용하는 응용 프로그램에서 링크 어 그리 게이션 기능을 가진 스위칭 허브 응용 프로그램을 실현하고 있습니다.

5.4 정리

이 장에서는 링크 통합 라이브러리 사용을 주제로 다음 항목 대해 설명했습니다.

- 「_CONTEXTS」을 이용한 라이브러리 사용 방법
- 사용자 정의 이벤트를 정의하는 방법과 이벤트 트리거의 발생 방법

스패닝 트리

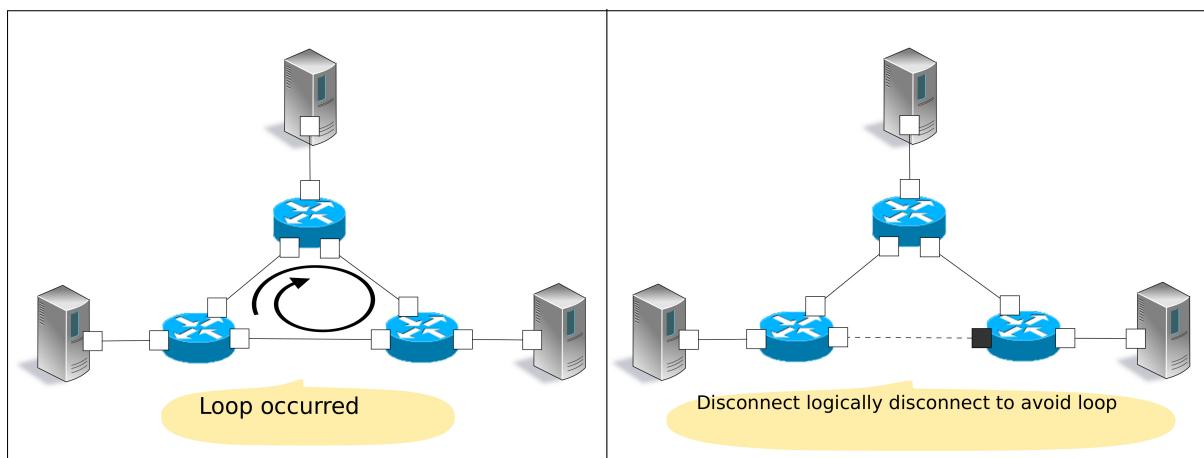
이 장에서는 Ryu를 이용한 스패닝 트리의 구현 방법을 설명하고 있습니다.

6.1 스패닝 트리

스패닝 트리 루프 구조를 가지는 네트워크의 브로드 캐스트 스톰의 발생을 억제하는 기능입니다. 또한 루프를 방지한다는 본래의 기능을 응용하여 네트워크 고장이 발생했을 때 자동으로 경로를 전환 네트워크 중복 보장의 수단으로도 이용됩니다.

스패닝 트리에는 STP, RSTP, PVST + MSTP 등 여러가지 종류가 있습니다만, 이 장에서는 가장 기본적인 STP 구현을 살펴 보겠습니다.

STP (spanning tree protocol : IEEE 802.1D) 네트워크를 논리적 트리로 취급하고, 각 스위치 (장에서는 브리지이라고 부를 수 있습니다) 포트를 프레임 전송 가능 또는 불가능 상태로 설정하는 것으로, 루프 구조를 가진 네트워크에서 브로드 캐스트 폭풍의 발생을 억제합니다.



STP는 브리지간에 BPDU (Bridge Protocol Data Unit) 패킷을 상호 교환하고 브리지와 포트 정보를 비교함으로서, 각 포트의 프레임 전송 여부를 결정합니다.

구체적으로는 다음과 같은 순서에 의해 실현됩니다.

1. 루트 브리지의 선출

브리지 사이의 BPDU 패킷 교환을 통해 최소의 브리지 ID를 갖는 브리지 루트 브리지로 선출 됩니다. 이후는 루트 브리지 만 원래 BPDU 패킷을 전송하고 다른 브리지가 루트 브리지에서 수신 한 BPDU 패킷을 전송합니다.

주석: 브리지 ID는 각 브리지에 설정된 브리지 priority 와 특정 포트의 MAC 주소의 조합으로 산출됩니다.

브리지 ID

상위2byte	하위6byte
브리지priority	MAC주소

2. 포트의 역할 결정

각 포트의 루트 브리지까지의 비용을 바탕으로, 포트의 역할을 결정합니다.

- 루트 포트 (Root port)

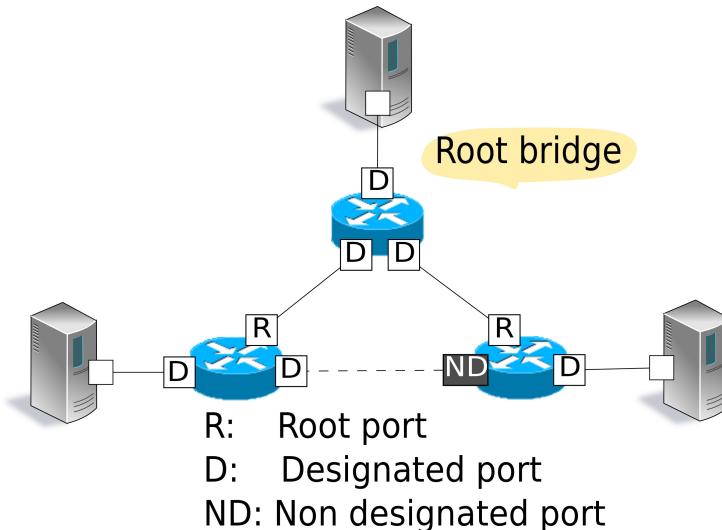
브리지에서 가장 루트 브리지까지의 비용이 작은 포트. 루트 브리지에서 BPDU 패킷을 수신하는 포트입니다.

- 지정 포트 (Designated port)

각 링크의 루트 브리지까지의 비용이 작은 쪽의 포트. 루트 브리지로부터 받은 BPDU 패킷을 전송하는 포트입니다. 루트 브리지의 포트는 모두 지정된 포트입니다.

- 비지정 포트 (Non designated port)

루트 포트 지정 포트 이외의 포트. 프레임 전송을 억제하는 포트입니다.



주석: 루트 브리지까지의 비용은 각 포트에서 수신한 BPDU 패킷 설정에서 다음과 같이 비교됩니다.

우선1:root path cost 값의 비교

각 브리지는 BPDU 패킷을 전송할 때 출력 포트에 설정된 path cost 값을 BPDU 패킷의 root path cost 값에 더합니다. 이렇게 하면 root path cost 값은 루트 브리지에 도달 할 때까지 통해 각 링크의 path cost 값의 합계입니다.

우선2:root path cost 값이 같으면 대향 브리지의 브리지 ID별로 비교.

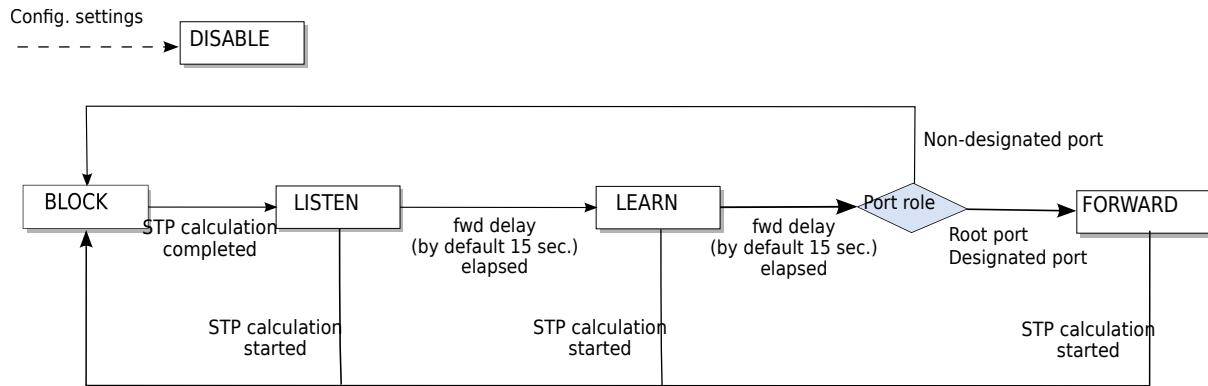
우선3:대향 브리지의 브리지 ID가 같은 경우 (각 포트가 동일한 브리지에 연결된 케이스), 대향 포트의 포트 ID별로 비교.

포트 ID

상위2byte	하위2byte
포트 priority	포트 번호

3. 포트의 상태 전이

포트 역할 결정 후 (STP 계산의 완료시) 각 포트는 LISTEN 상태입니다. 그런 다음에 나타내는 상태 전환을 실시해, 최종적으로 각 포트의 역할에 따라 FORWARD 상태 또는 BLOCK 상태로 전환합니다. 구성에서 사용 할 포트 및 설정 된 포트는 DISABLE 상태가되고, 이후 상태 전환되지 않습니다.



이러한 작업이 각 브리지에서 실행되는 것으로, 프레임 전송하는 포트와 프레임 전송을 억제하는 포트를 결정하고 네트워크의 루프가 해소됩니다.

또한 링크 다운 및 BPDU 패킷 max age (기본 20 초) 사이의 미수신에 의한 고장 감지 또는 포트의 추가 등에 의해 네트워크 토플로지 변경 내용 발견 한 경우 각 브리지에서 위의 1. 2. 3.을 실행 트리의 재구성 됩니다 (STP 재 계산).

6.2 Ryu 응용 프로그램의 실행

스페닝 트리 기능을 OpenFlow를 이용하여 실현 한 Ryu 스페닝 트리 응용 프로그램을 실행 해 봅니다.

Ryu 소스 트리에 포함되어 있는 simple_switch_stp.py는 OpenFlow 1.0 전용 응용 프로그램이기 때문에 여기에서는 새롭게 OpenFlow 1.3에 대응 한 simple_switch_stp_13.py를 만들합니다. 이 프로그램은 「[스위칭 허브](#)」스페닝 트리 기능을 추가 한 응용 프로그램입니다.

소스 이름: simple_switch_stp_13.py

```

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import dpid as dpid_lib
from ryu.lib import stplib
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.stp = kwargs['stplib']

        # Sample of stplib config.
        # please refer to stplib.Stp.set_config() for details.
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                  {'bridge': {'priority': 0x8000}},
                  dpid_lib.str_to_dpid('0000000000000002'):
                  {'bridge': {'priority': 0x9000}},
                  dpid_lib.str_to_dpid('0000000000000003'):
                  {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
  
```

```

def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # install table-miss flow entry
    #
    # We specify NO_BUFFER to max_len of the output action due to
    # OVS bug. At this moment, if we specify a lesser number, e.g.,
    # 128, OVS will send Packet-In with invalid buffer_id and
    # truncated packet data. In that case, we cannot output packets
    # correctly.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER) ]
    self.add_flow(datapath, 0, match, actions)

def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPIstructionActions(ofproto.OFPI_APPLY_ACTIONS,
                                         actions) ]

    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                           match=match, instructions=inst)
    datapath.send_msg(mod)

def delete_flow(self, datapath):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    for dst in self.mac_to_port[datapath.id].keys():
        match = parser.OFPMatch(eth_dst=dst)
        mod = parser.OFPFlowMod(
            datapath, command=ofproto.OFPFC_DELETE,
            out_port=ofproto.OFPP_ANY, out_group=ofproto.OFGP_ANY,
            priority=1, match=match)
        datapath.send_msg(mod)

@set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ether.ether)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:

```

```

        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                             in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)

@set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
def _topology_change_handler(self, ev):
    dp = ev.dp
    dpid_str = dpid_lib.dpid_to_str(dp.id)
    msg = 'Receive topology change event. Flush MAC table.'
    self.logger.debug("[dpid=%s] %s", dpid_str, msg)

    if dp.id in self.mac_to_port:
        self.delete_flow(dp)
        del self.mac_to_port[dp.id]

@set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
def _port_state_change_handler(self, ev):
    dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
    of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                stplib.PORT_STATE_BLOCK: 'BLOCK',
                stplib.PORT_STATE_LISTEN: 'LISTEN',
                stplib.PORT_STATE_LEARN: 'LEARN',
                stplib.PORT_STATE_FORWARD: 'FORWARD'}
    self.logger.debug("[dpid=%s] [port=%d] state=%s",
                     dpid_str, ev.port_no, of_state[ev.port_state])

```

6.2.1 실험 환경 구축

스패닝 트리 응용 프로그램의 동작 확인을 할 실험 환경을 구축합니다.

VM 이미지 사용을 위한 환경 설정 및 로그인 방법 등은 「[스위칭 허브](#)」을 참조하십시오.

루프 구조를 가지는 특수한 위상으로 작동시키기 위해 「[링크 어그리게이션](#)」 뿐만 아니라 토플로지 구축 스크립트는 mininet 환경을 구축합니다.

소스 이름: spanning_tree.py

```

#!/usr/bin/env python

from mininet.cli import CLI
from mininet.link import Link
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.term import makeTerm

if '__main__' == __name__:
    net = Mininet(controller=RemoteController)

```

```

c0 = net.addController('c0')

s1 = net.addSwitch('s1')
s2 = net.addSwitch('s2')
s3 = net.addSwitch('s3')

h1 = net.addHost('h1')
h2 = net.addHost('h2')
h3 = net.addHost('h3')

Link(s1, h1)
Link(s2, h2)
Link(s3, h3)

Link(s1, s2)
Link(s2, s3)
Link(s3, s1)

net.build()
c0.start()
s1.start([c0])
s2.start([c0])
s3.start([c0])

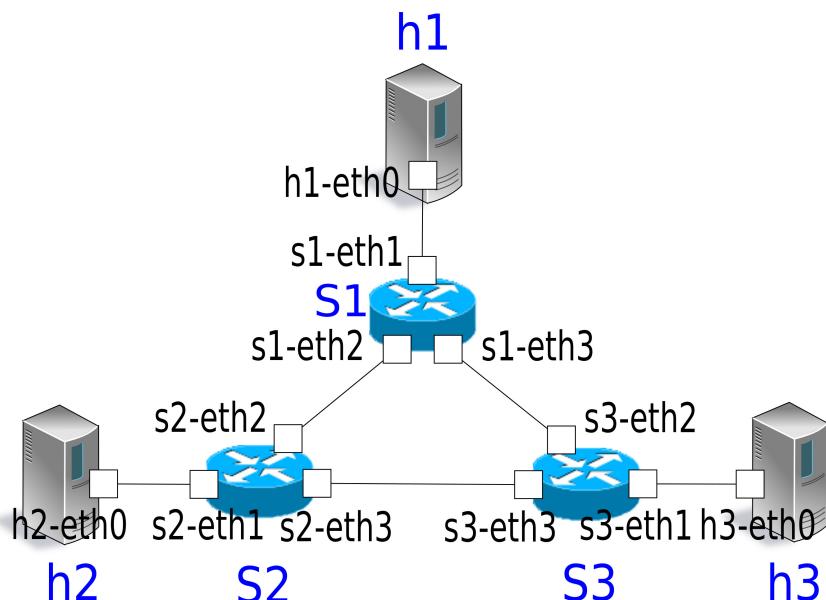
net.terms.append(makeTerm(c0))
net.terms.append(makeTerm(s1))
net.terms.append(makeTerm(s2))
net.terms.append(makeTerm(s3))
net.terms.append(makeTerm(h1))
net.terms.append(makeTerm(h2))
net.terms.append(makeTerm(h3))

CLI(net)

net.stop()

```

VM 환경에서 이 프로그램을 실행하면 스위치 s1, s2, s3 사이에서 루프가 존재하는 토플로지가 됩니다.



net 명령의 실행 결과는 다음과 같습니다.

```
ryu@ryu-vm:~$ sudo ./spanning_tree.py
Unable to contact the remote controller at 127.0.0.1:6633
mininet> net
c0
s1 lo: s1-eth1:h1-eth0 s1-eth2:s2-eth2 s1-eth3:s3-eth3
s2 lo: s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo: s3-eth1:h3-eth0 s3-eth2:s2-eth3 s3-eth3:s1-eth3
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
```

6.2.2 OpenFlow 버전 설정

사용하는 OpenFlow 버전을 1.3으로 설정합니다. 이 명령 입력 스위치 s1, s2, s3의 xterm에서 실시해주세요.

Node: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

Node: s2:

```
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

Node: s3:

```
root@ryu-vm:~# ovs-vsctl set Bridge s3 protocols=OpenFlow13
```

6.2.3 스위칭 허브의 실행

준비 하였으므로 Ryu 응용 프로그램을 실행합니다. 윈도우 제목이 「Node: c0 (root)」이다 xterm에서 다음 명령을 실행합니다.

Node: c0:

```
root@ryu-vm:~$ ryu-manager ./simple_switch_stp_13.py
loading app simple_switch_stp_13.py
loading app ryu.controller.ofp_handler
loading app ryu.controller.ofp_handler
instantiating app None of Stp
creating context stplib
instantiating app simple_switch_stp_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
```

OpenFlow 스위치 시작 STP 계산

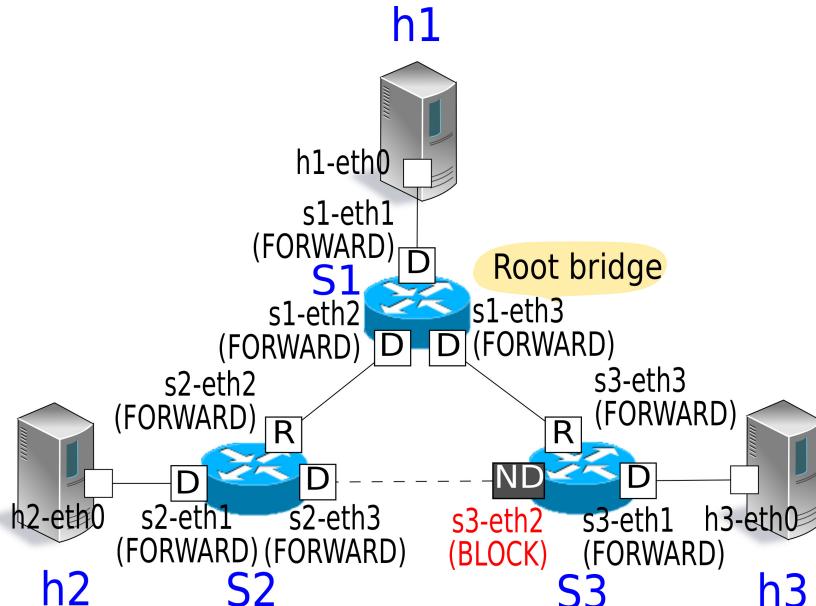
각 OpenFlow 스위치와 컨트롤러의 연결이 완료되면 BPDU 패킷의 교환이 시작 루트 브리지 선출 포트 역할 설정 포트 상태 천이가 이루어집니다.

```
[STP] [INFO] dpid=0000000000000001: Join as stp bridge.
[STP] [INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000002: Join as stp bridge.
[STP] [INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000001: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
```



```
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000002: [port=2] ROOT_PORT          / FORWARD
[STP] [INFO] dpid=00000000000000002: [port=3] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=3] ROOT_PORT          / FORWARD
[STP] [INFO] dpid=00000000000000001: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT      / FORWARD
```

이 결과, 최종적으로 각 포트는 FORWARD 상태 또는 BLOCK 상태입니다.



패킷이 루프하지 않는다는 것을 확인하기 위해 호스트 1에서 호스트 2로 ping을 실행합니다.

ping 명령을 실행하기 전에 tcpdump 명령을 실행해야합니다.

Node: s1:

```
root@ryu-vm:~# tcpdump -i s1-eth2 arp
```

Node: s2:

```
root@ryu-vm:~# tcpdump -i s2-eth2 arp
```

Node: s3:

```
root@ryu-vm:~# tcpdump -i s3-eth2 arp
```

토플로지 작성 스크립트를 실행 한 콘솔에서 다음 명령을 실행 호스트 1에서 호스트 2로 ping을 실행합니다.

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=84.4 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.657 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.074 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.054 ms
64 bytes from 10.0.0.2: icmp_req=6 ttl=64 time=0.053 ms
64 bytes from 10.0.0.2: icmp_req=7 ttl=64 time=0.041 ms
64 bytes from 10.0.0.2: icmp_req=8 ttl=64 time=0.049 ms
64 bytes from 10.0.0.2: icmp_req=9 ttl=64 time=0.074 ms
```

```
64 bytes from 10.0.0.2: icmp_req=10 ttl=64 time=0.073 ms
64 bytes from 10.0.0.2: icmp_req=11 ttl=64 time=0.068 ms
^C
--- 10.0.0.2 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 9998ms
rtt min/avg/max/mdev = 0.041/7.784/84.407/24.230 ms
```

tcpdump의 출력에서 ARP 루프하지 않을 수 확인할 수 있습니다.

Node: s1:

```
root@ryu-vm:~# tcpdump -i s1-eth2 arp
tcpdump: WARNING: s1-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1-eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
11:30:24.692797 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
11:30:24.749153 ARP, Reply 10.0.0.2 is-at 82:c9:d7:e9:b7:52 (oui Unknown), length 28
11:30:29.797665 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
11:30:29.798250 ARP, Reply 10.0.0.1 is-at c2:a4:54:83:43:fa (oui Unknown), length 28
```

Node: s2:

```
root@ryu-vm:~# tcpdump -i s2-eth2 arp
tcpdump: WARNING: s2-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s2-eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
11:30:24.692824 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
11:30:24.749116 ARP, Reply 10.0.0.2 is-at 82:c9:d7:e9:b7:52 (oui Unknown), length 28
11:30:29.797659 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
11:30:29.798254 ARP, Reply 10.0.0.1 is-at c2:a4:54:83:43:fa (oui Unknown), length 28
```

Node: s3:

```
root@ryu-vm:~# tcpdump -i s3-eth2 arp
tcpdump: WARNING: s3-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s3-eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
11:30:24.698477 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
```

감지 된 고장 STP 재 계산

다음 링크 다운이 일어 났을 때의 STP 재 계산의 동작을 확인합니다. 각 OpenFlow 스위치 시작 후 STP 계산이 완료된 상태에서 다음 명령을 실행하여 포트를 다운시킵니다.

Node: s2:

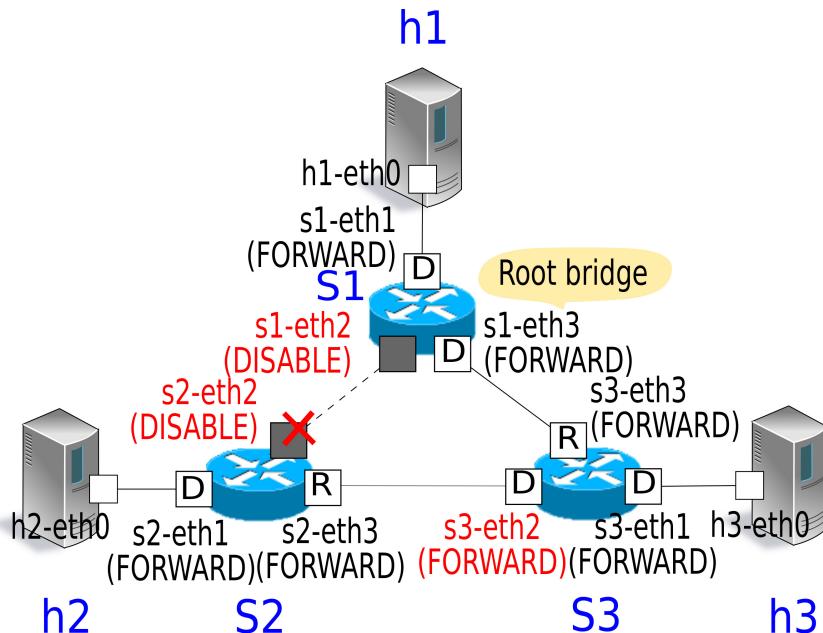
```
root@ryu-vm:~# ifconfig s2-eth2 down
```

링크 다운이 감지되고 STP 재 계산이 실행됩니다.

```
[STP] [INFO] dpid=0000000000000002: [port=2] Link down.
[STP] [INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT / DISABLE
[STP] [INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT / BLOCK
[STP] [INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT / BLOCK
[STP] [INFO] dpid=0000000000000002: Root bridge.
[STP] [INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT / LISTEN
[STP] [INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT / LISTEN
[STP] [INFO] dpid=0000000000000001: [port=2] Link down.
[STP] [INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT / DISABLE
[STP] [INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=0000000000000003: [port=2] Wait BPDU timer is exceeded.
[STP] [INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT / BLOCK
```

```
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: Root bridge.
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=3] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: Non root bridge.
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=3] ROOT_PORT          / LISTEN
[STP] [INFO] dpid=00000000000000002: [port=3] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000002: Non root bridge.
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000002: [port=3] ROOT_PORT          / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000003: [port=3] ROOT_PORT          / LEARN
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000002: [port=3] ROOT_PORT          / LEARN
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000003: [port=3] ROOT_PORT          / FORWARD
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000002: [port=3] ROOT_PORT          / FORWARD
```

지금까지 BLOCK 상태였다 s3-eth2 포트가 FORWARD 상태가 되고, 다시 프레임 전송 가능한 상태가 된 것을 확인할 수 있습니다.



고장 복구시 STP 재 계산

계속 링크 다운이 재개 될 때의 STP 재 계산의 동작을 확인합니다. 링크 다운 된 상태에서 다음 명령을 실행하여 포트를 활성화시킵니다.

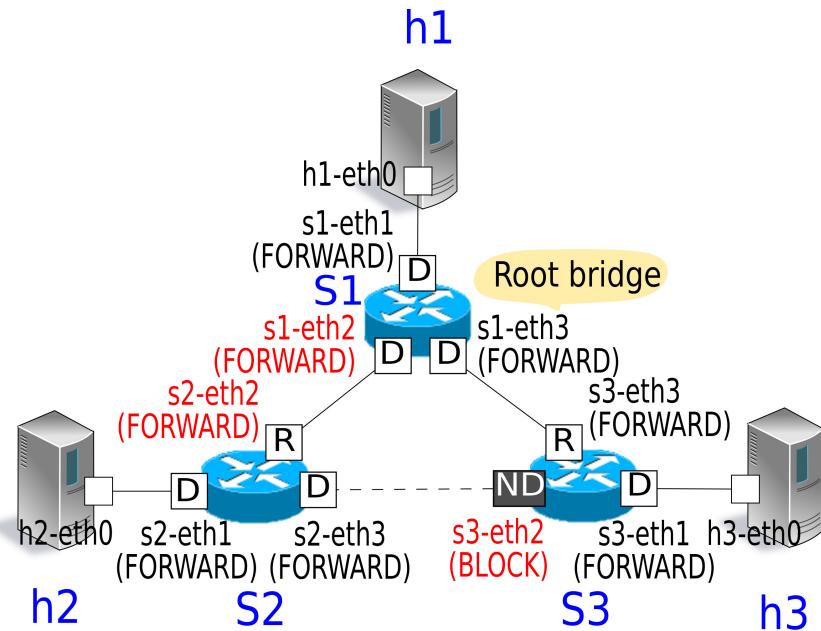
Node: s2:

```
root@ryu-vm:~# ifconfig s2-eth2 up
```

링크 복구가 감지되고 STP 재 계산이 실행됩니다.

```
[STP] [INFO] dpid=00000000000000000002: [port=2] Link down.
[STP] [INFO] dpid=00000000000000000002: [port=2] DESIGNATED_PORT      / DISABLE
[STP] [INFO] dpid=00000000000000000002: [port=2] Link up.
[STP] [INFO] dpid=00000000000000000002: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000001: [port=2] Link up.
[STP] [INFO] dpid=00000000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000001: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000001: Root bridge.
[STP] [INFO] dpid=00000000000000000001: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000001: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000002: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000002: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000002: Non root bridge.
[STP] [INFO] dpid=00000000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000002: [port=2] ROOT_PORT          / LISTEN
[STP] [INFO] dpid=00000000000000000002: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000003: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000000003: Non root bridge.
[STP] [INFO] dpid=00000000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000000003: [port=2] NON_DESIGNATED_PORT / LISTEN
[STP] [INFO] dpid=00000000000000000003: [port=3] ROOT_PORT          / LISTEN
[STP] [INFO] dpid=00000000000000000001: [port=1] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000001: [port=2] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000001: [port=3] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000002: [port=2] ROOT_PORT          / LEARN
[STP] [INFO] dpid=00000000000000000002: [port=3] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000002: [port=3] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000003: [port=1] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=00000000000000000003: [port=2] NON_DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000000003: [port=3] ROOT_PORT          / LEARN
[STP] [INFO] dpid=00000000000000000001: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000000001: [port=2] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000000001: [port=3] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000000002: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000000002: [port=2] ROOT_PORT          / FORWARD
[STP] [INFO] dpid=00000000000000000002: [port=3] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=00000000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP] [INFO] dpid=00000000000000000003: [port=3] ROOT_PORT          / FORWARD
```

응용 프로그램 시작시와 같은 트리 구성이 다시 프레임 전송 가능 상태가 된 것을 확인할 수 있습니다.



6.3 OpenFlow 스패닝 트리

Ryu 스패닝 트리 응용 프로그램에서 OpenFlow를 사용하여 어떻게 스패닝 트리 기능을 수행하고 있는지를 살펴 보겠습니다.

OpenFlow 1.3에는 다음과 같은 포트의 동작을 설정하는 구성이 준비되어 있습니다. Port Modification 메시지를 OpenFlow 스위치에 게시 하여 포트의 프레임 전송 여부 등의 동작을 제어 할 수 있습니다.

값	설명
OFPPC_PORT_DOWN	유지 보수에 의해 무효 설정된 상태입니다
OFPPC_NO_RECV	해당 포트에서 수신 한 모든 패킷을 폐기합니다
OFPPC_NO_FWD	해당 포트에서 패킷 전송하지 않습니다
OFPPC_NO_PACKET_IN	table-miss이 경우 Packet-In 메시지를 보내지 않습니다

또한, 포트 당 BPDU 패킷 수신 및 BPDU 이외의 패킷 수신을 제어하기 위해 BPDU 패킷을 Packet-In 항목 및 BPDU 이외의 패킷을 drop하기 항목을 각각 Flow Mod 메시지는 OpenFlow 스위치에 등록합니다.

컨트롤러는 각 OpenFlow 스위치에 대해 아래와 같이 포트 구성 설정 및 항목 설정함으로써 포트 상태에 따라 BPDU 패킷 송수신 또는 MAC 주소 학습 (BPDU 이외의 패킷 수신) 프레임 전송 (BPDU 이외의 패킷 전송) 제어합니다.

상태	포트 구성	항목
DISABLE	NO_RECV / NO_FWD	설정 없음
BLOCK	NO_FWD	BPDU Packet-In / BPDU 이외 drop
LISTEN	설정 없음	BPDU Packet-In / BPDU 이외 drop
LEARN	설정 없음	BPDU Packet-In / BPDU 이외 drop
FORWARD	설정 없음	BPDU Packet-In

주석: Ryu에 구현 된 스패닝 트리의 라이브러리는 편의상 LEARN 상태에서 MAC 주소 학습 (BPDU 이외의 패킷 수신)을 실시하고 있지 않습니다.

이러한 설정뿐 아니라 컨트롤러는 OpenFlow 스위치와 연결할 때 수집 한 포트 정보와 각 OpenFlow 스위치가 받은 BPDU 패킷에 설정된 루트 브리지 정보를 바탕으로 보내기 위한 BPDU 패킷을 구축 Packet-Out 메시지를 발행하는 것으로, OpenFlow 스위치 사이의 BPDU 패킷의 교환을 제공합니다.

6.4 Ryu 스패닝 트리 구현

이어 Ryu를 사용하여 구현 된 스패닝 트리의 소스 코드를 살펴 보겠습니다. 스패닝 트리의 소스 코드, Ryu 소스 트리에 있습니다.

ryu/lib/stplib.py

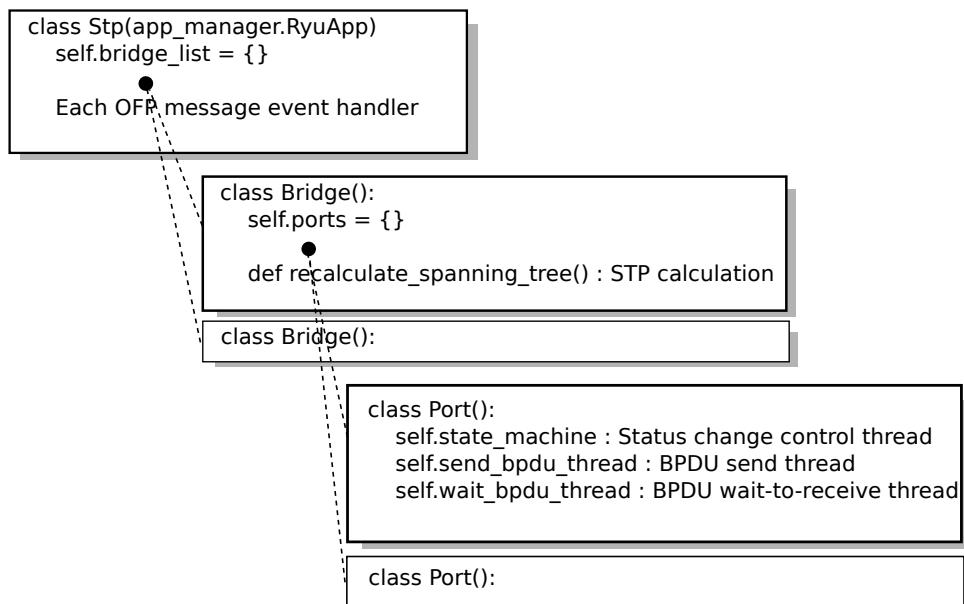
ryu/app/simple_switch_stp.py

stplib.py는 BPDU 패킷 교환이나 각 포트의 역할 · 상태 관리 등의 스패닝 트리 기능을 제공하는 라이브러리입니다. simple_switch_stp.py는 스패닝 트리 라이브러리를 적용하여 스위칭 허브의 응용 프로그램에 스패닝 트리 기능을 추가 한 응용 프로그램입니다.

주의: simple_switch_stp.py는 OpenFlow 1.0 전용 응용 프로그램 이기 때문에 이 장에서는 「Ryu 응용 프로그램의 실행」으로 보여 주었다 OpenFlow 1.3에 대응 한 simple_switch_stp_13.py 기반으로 응용 프로그램 자세한 내용을 설명합니다.

6.4.1 라이브러리의 구현

라이브러리 개요



STP 라이브러리 (`Stp` 클래스 인스턴스)가 OpenFlow 스위치 컨트롤러 연결을 감지하면 `Bridge` 클래스 인스턴스 `Port` 클래스 인스턴스가 생성됩니다. 각 클래스 인스턴스가 생성 · 시작 된 후에는

- `Stp` 클래스 인스턴스에서 OpenFlow 메시지 수신 알림
- `Bridge` 클래스 인스턴스의 STP 계산(루트 브리지 선택 및 각 포트의 역할 선택)
- `Port` 클래스 인스턴스의 포트 상태 전이 · BPDU 패킷 전송

가 연동 스패닝 트리 기능을 제공합니다.

구성 설정 항목

STP 라이브러리는 `Stp.set_config()` 메소드에 의해 브리지 포트 구성 설정 IF를 제공합니다. 설정 가능한 항목은 다음과 같습니다.

- bridge

항목	설명	기본값
priority	브리지 우선 순위	0x8000
sys_ext_id	VLAN-ID 설정 (* 현재 STP 라이브러리는 VLAN 비 인식)	0
max_age	BPDU 패킷의 수신 타이머 값	20[sec]
hello_time	BPDU 패킷의 전송 간격	2 [sec]
fwd_delay	각 포트가 LISTEN 상태 및 LEARN 상태에 머무는 시간	15[sec]

- port

항목	설명	기본값
priority	포트 우선 순위	0x80
path_cost	링크의 비용 값 링크	속도를 바탕으로 자동 설정
enable	포트 활성화/비활성화 설정	True

BPDU 패킷 전송

BPDU 패킷은 Port 클래스의 BPDU 패킷 전송 스레드 (Port.send_bpdu_thread)에서 실시하고 있습니다. 포트 역할이 지정 포트 (DESIGNATED_PORT)의 경우 루트 브리지에서 공지 된 hello time (Port.port_times.hello_time : 기본 2 초) 간격으로 BPDU 패킷 생성 (Port._generate_config_bpdu ()) 및 BPDU 패킷 전송 (Port.ofctl.send_packet_out ())합니다.

```
class Port(object):

    def __init__(self, dp, logger, config, send_ev_func, timeout_func,
                 topology_change_func, bridge_id, bridge_times, ofport):
        super(Port, self).__init__()

        # ...

        # BPDU handling threads
        self.send_bpdu_thread = PortThread(self._transmit_bpdu)

        # ...

    def _transmit_bpdu(self):
        while True:
            # Send config BPDU packet if port role is DESIGNATED_PORT.
            if self.role == DESIGNATED_PORT:

                # ...

                bpdu_data = self._generate_config_bpdu(flags)
                self.ofctl.send_packet_out(self.ofport.port_no, bpdu_data)

                # ...

            hub.sleep(self.port_times.hello_time)
```

보내는 BPDU 패킷은 OpenFlow 스위치 컨트롤러 연결시에 수집 한 포트 정보 (Port.ofport)이나 받은 BPDU 패킷에 설정된 루트 브리지 정보 (Port.port_priority, Port.port_times) 등을 바탕으로 구축됩니다.

```
class Port(object):

    def _generate_config_bpdu(self, flags):
        src_mac = self.ofport.hw_addr
        dst_mac = bpdu.BRIDGE_GROUP_ADDRESS
        length = (bpdu.bpdu._PACK_LEN + bpdu.ConfigurationBPDUs.PACK_LEN
                  + llc.llc._PACK_LEN + llc.ControlFormatU._PACK_LEN)

        e = ethernet.ethernet(dst_mac, src_mac, length)
```

```

l = llc.llc(llc.SAP_BPDU, llc.SAP_BPDU, llc.ControlFormatU())
b = bpdu.ConfigurationBPDUs(
    flags=flags,
    root_priority=self.port_priority.root_id.priority,
    root_mac_address=self.port_priority.root_id.mac_addr,
    root_path_cost=self.port_priority.root_path_cost+self.path_cost,
    bridge_priority=self.bridge_id.priority,
    bridge_mac_address=self.bridge_id.mac_addr,
    port_priority=self.port_id.priority,
    port_number=self.ofport.port_no,
    message_age=self.port_times.message_age+1,
    max_age=self.port_times.max_age,
    hello_time=self.port_times.hello_time,
    forward_delay=self.port_times.forward_delay)

pkt = packet.Packet()
pkt.add_protocol(e)
pkt.add_protocol(l)
pkt.add_protocol(b)
pkt.serialize()

return pkt.data

```

BPDU 패킷 수신

BPDU 패킷의 수신은 Stp 클래스의 Packet-In 이벤트 핸들러에 의해 감지되고 Bridge 클래스 인스턴스를 통해 Port 클래스 인스턴스에 통지됩니다. 이벤트 처리기 구현 「[스위칭 허브](#)」을 참조하십시오.

BPDU 패킷을 수신 한 포트는 이전에 수신 한 BPDU 패킷 이번받은 BPDU 패킷의 브리지 ID 등의 비교 (Stp.compare_bpdu_info ())를 실시해, STP 재 계산의 필요 여부 판정합니다. 이전에 수신 한 BPDU 보다 뛰어난 BPDU (SUPERIOR)를 받은 경우 “새로운 루트 브리지가 추가 된 ‘등의 네트워크 토플로지 변경이 발생했다는 것을 의미하기 위해 STP 재 계산의 계기가 됩니다.

class Port(object):

```

def recv_config_bpdu(self, bpdu_pkt):
    # Check received BPDU is superior to currently held BPDU.
    root_id = BridgeId(bpdu_pkt.root_priority,
                        bpdu_pkt.root_system_id_extension,
                        bpdu_pkt.root_mac_address)
    root_path_cost = bpdu_pkt.root_path_cost
    designated_bridge_id = BridgeId(bpdu_pkt.bridge_priority,
                                     bpdu_pkt.bridge_system_id_extension,
                                     bpdu_pkt.bridge_mac_address)
    designated_port_id = PortId(bpdu_pkt.port_priority,
                                bpdu_pkt.port_number)

    msg_priority = Priority(root_id, root_path_cost,
                           designated_bridge_id,
                           designated_port_id)
    msg_times = Times(bpdu_pkt.message_age,
                      bpdu_pkt.max_age,
                      bpdu_pkt.hello_time,
                      bpdu_pkt.forward_delay)

    rcv_info = Stp.compare_bpdu_info(self.designated_priority,
                                    self.designated_times,
                                    msg_priority, msg_times)

    # ...

return rcv_info, rcv_tc

```

고장 검출

링크 단 등의 직접 고장이나 일정 시간 루트 브리지에서 BPDU 패킷을 받을 수 없는 간접 고장을 검출 한 경우에도 STP 재 계산의 계기가 됩니다.

링크 차단은 Stp 클래스의 PortStatus 이벤트 핸들러에 의해 감지하고 Bridge 클래스 인스턴스에 통지됩니다.

BPDU 패킷의 수신 시간 제한은 Port 클래스의 BPDU 패킷 수신 스레드 (Port.wait_bpdu_thread)에서 검색합니다. max age (기본 20 초) 동안 루트 브리지에서 BPDU 패킷을 수신 할 수 없는 경우 간접 고장이라고 판단하고 Bridge 클래스 인스턴스에 통지됩니다.

BPDU 수신 타이머 업데이트와 시간 초과를 감지에는 hub 모듈 (ryu.lib.hub) 의 hub.Event 와 hub.Timeout 을 사용합니다. “ hub.Event ” 는 hub.Event.wait () 에서 wait 상태에 들어 hub.Event.set () 가 실행될 때까지 스레드가 중단됩니다. hub.Timeout 는 지정된 제한 시간 내에 try 절이 처리가 종료하지 않으면 hub.Timeout 예외를 발생합니다. hub.Event 가 wait 상태에 들어 hub.Timeout 에 지정된 제한 시간 내에 hub.Event.set () 가 실행되지 않을 때 BPDU 패킷의 수신 시간 초과 판단 Bridge 클래스의 STP 재 계산 프로세스를 호출합니다.

```
class Port(object):

    def __init__(self, dp, logger, config, send_ev_func, timeout_func,
                 topology_change_func, bridge_id, bridge_times, ofport):
        super(Port, self).__init__()
        # ...
        self.wait_bpdu_timeout = timeout_func
        # ...
        self.wait_bpdu_thread = PortThread(self._wait_bpdu_timer)

    # ...

    def _wait_bpdu_timer(self):
        time_exceed = False

        while True:
            self.wait_timer_event = hub.Event()
            message_age = (self.designated_times.message_age
                           if self.designated_times else 0)
            timer = self.port_times.max_age - message_age
            timeout = hub.Timeout(timer)
            try:
                self.wait_timer_event.wait()
            except hub.Timeout as t:
                if t is not timeout:
                    err_msg = 'Internal error. Not my timeout.'
                    raise RyuException(msg=err_msg)
                self.logger.info('[port=%d] Wait BPDU timer is exceeded.',
                                self.ofport.port_no, extra=self.dpid_str)
                time_exceed = True
            finally:
                timeout.cancel()
                self.wait_timer_event = None

        if time_exceed:
            break

    if time_exceed: # Bridge.recalculate_spanning_tree
        hub.spawn(self.wait_bpdu_timeout)
```

받은 BPDU 패킷의 비교 처리 (Stp.compare_bpdu_info ())에 의해 SUPERIOR 또는 REPEATED

판정 된 경우는 루트 브리지에서 BPDU 패킷을 수신 할 수 있는 것을 의미하기 때문에 BPDU 수신 타이머 업데이트 (Port._update_wait_bpdu_timer())합니다. hub.Event'의 딕셔너리 Port.wait_timer_event 의 set() 처리에 의해 Port.wait_timer_event 는 wait 상태에서 해방 된 BPDU 패킷 수신 스레드 (Port.wait_bpdu_thread) 는 except hub.Timeout 절 시간 제한 처리에 들어 가지 않고도 타이머를 취소하고 다시 타이머를 세트 다시하는 것으로 다음의 BPDU 패킷 수신 을 시작합니다.

```
class Port(object):

    def recv_config_bpdu(self, bpdu_pkt):
        # ...

        rcv_info = Stp.compare_bpdu_info(self.designated_priority,
                                         self.designated_times,
                                         msg_priority, msg_times)
        # ...

        if ((rcv_info is SUPERIOR or rcv_info is REPEATED)
            and (self.role is ROOT_PORT
                  or self.role is NON_DESIGNATED_PORT)):
            self._update_wait_bpdu_timer()

        # ...

    def _update_wait_bpdu_timer(self):
        if self.wait_timer_event is not None:
            self.wait_timer_event.set()
            self.wait_timer_event = None
```

STP계산

STP 계산(루트 브리지 선택 및 각 포트의 역할 선택)은 Bridge 클래스에서 실행합니다.

STP 계산을 수행하는 경우에는 네트워크 토폴로지 변경이 발생하고 패킷이 루프 할 수 있기 때문에 일단 모든 포트를 BLOCK 상태로 설정 (port.down)하고 또한 토폴로지 변경 이벤트 (EventTopologyChange) 상위 APL에게 통지함으로써 학습 된 MAC 주소 정보의 초기화를 재촉합니다.

그럼 Bridge._spanning_tree_algorithm() 루트 브리지 및 포트 역할을 선택한 다음, 각 포트를 LISTEN 상태에서 시작 (port.up)하여 포트 상태 전이를 시작합니다.

```
class Bridge(object):

    def recalculate_spanning_tree(self, init=True):
        """ Re-calculation of spanning tree. """
        # All port down.
        for port in self.ports.values():
            if port.state is not PORT_STATE_DISABLE:
                port.down(PORT_STATE_BLOCK, msg_init=init)

        # Send topology change event.
        if init:
            self.send_event(EventTopologyChange(self.dp))

        # Update tree roles.
        port_roles = {}
        self.root_priority = Priority(self.bridge_id, 0, None, None)
        self.root_times = self.bridge_times

        if init:
            self.logger.info('Root bridge.', extra=self.dpid_str)
            for port_no in self.ports.keys():
```

```

        port_roles[port_no] = DESIGNATED_PORT
else:
    (port_roles,
     self.root_priority,
     self.root_times) = self._spanning_tree_algorithm()

    # All port up.
    for port_no, role in port_roles.items():
        if self.ports[port_no].state is not PORT_STATE_DISABLE:
            self.ports[port_no].up(role, self.root_priority,
                                  self.root_times)

```

루트 브리지의 선출을 위해 브리지 ID 등 자신의 브리지 정보 각 포트에서 수신 한 BPDU 패킷에 설정된 다른 브리지 정보를 비교합니다(Bridge._select_root_port).

이 결과, 루트 포트가 발견되면 (자신의 브리지 정보보다 포트가 받은 다른 브리지 정보가 출중 한 경우) 다른 브리지가 루트 브리지다고 판단 지정 포트의 선출(Bridge._select_designated_port)과 비지정 포트 선출(루트 포트 / 지정 포트 이외의 포트를 비지정 포트로 선출)합니다.

한편, 루트 포트가 없는 경우 (자신의 브리지 정보가 가장 우수했다 경우) 자신을 루트 브리지 판단 각 포트는 모두 지정된 포트입니다.

```

class Bridge(object):

    def _spanning_tree_algorithm(self):
        """ Update tree roles.
        - Root bridge:
          all port is DESIGNATED_PORT.
        - Non root bridge:
          select one ROOT_PORT and some DESIGNATED_PORT,
          and the other port is set to NON_DESIGNATED_PORT."""
        port_roles = {}

        root_port = self._select_root_port()

        if root_port is None:
            # My bridge is a root bridge.
            self.logger.info('Root bridge.', extra=self.dpid_str)
            root_priority = self.root_priority
            root_times = self.root_times

            for port_no in self.ports.keys():
                if self.ports[port_no].state is not PORT_STATE_DISABLE:
                    port_roles[port_no] = DESIGNATED_PORT
        else:
            # Other bridge is a root bridge.
            self.logger.info('Non root bridge.', extra=self.dpid_str)
            root_priority = root_port.designated_priority
            root_times = root_port.designated_times

            port_roles[root_port.ofport.port_no] = ROOT_PORT

            d_ports = self._select_designated_port(root_port)
            for port_no in d_ports:
                port_roles[port_no] = DESIGNATED_PORT

            for port in self.ports.values():
                if port.state is not PORT_STATE_DISABLE:
                    port_roles.setdefault(port.ofport.port_no,
                                         NON_DESIGNATED_PORT)

        return port_roles, root_priority, root_times

```

포트 상태 전이

포트의 상태 전환 처리, Port 클래스의 상태 전이 제어 스레드 (Port.state_machine) 에서 실행하고 있습니다. 다음 상태로 전환 될 때까지 타이머를 Port._get_timer() 에서 가져 타이머 만료 후 Port._get_next_state() 에서 다음 상태를 가져, 상태 전환합니다. 또한 STP 재 계산이 발생 지금 까지의 포트 상태에 관계없이 BLOCK 상태로 전환시키는 케이스 등 Port._change_status() 가 실행 된 경우 도 상태 천이가 이루어집니다. 이러한 작업은 「고장 검출」처럼 hub 모듈 hub.Event 와 hub.Timeout 을 이용하여 실현하고 있습니다.

```

class Port(object):

    def _state_machine(self):
        """ Port state machine.
            Change next status when timer is exceeded
            or _change_status() method is called."""

        # ...

        while True:
            self.logger.info('[port=%d] %s / %s', self.ofport.port_no,
                            role_str[self.role], state_str[self.state],
                            extra=self.dpid_str)

            self.state_event = hub.Event()
            timer = self._get_timer()
            if timer:
                timeout = hub.Timeout(timer)
                try:
                    self.state_event.wait()
                except hub.Timeout as t:
                    if t is not timeout:
                        err_msg = 'Internal error. Not my timeout.'
                        raise RyuException(msg=err_msg)
                    new_state = self._get_next_state()
                    self._change_status(new_state, thread_switch=False)
                finally:
                    timeout.cancel()
            else:
                self.state_event.wait()

            self.state_event = None

    def _get_timer(self):
        timer = {PORT_STATE_DISABLE: None,
                 PORT_STATE_BLOCK: None,
                 PORT_STATE_LISTEN: self.port_times.forward_delay,
                 PORT_STATE_LEARN: self.port_times.forward_delay,
                 PORT_STATE_FORWARD: None}
        return timer[self.state]

    def _get_next_state(self):
        next_state = {PORT_STATE_DISABLE: None,
                      PORT_STATE_BLOCK: None,
                      PORT_STATE_LISTEN: PORT_STATE_LEARN,
                      PORT_STATE_LEARN: (PORT_STATE_FORWARD
                                         if (self.role is ROOT_PORT or
                                              self.role is DESIGNATED_PORT)
                                         else PORT_STATE_BLOCK),
                      PORT_STATE_FORWARD: None}
        return next_state[self.state]

```

6.4.2 응용 프로그램 구현

「Ryu 응용 프로그램의 실행」에 나와있는 OpenFlow 1.3 대응의 스패닝 트리 응용 프로그램 (simple_switch_stp_13.py)과 「스위칭 허브」 스위칭 허브의 차이를 순서대로 설명하고 있습니다.

「_CONTEXTS」 설정

「링크 어그리게이션」과 같이 STP 라이브러리를 사용하는 CONTEXT를 등록합니다.

```
from ryu.lib import stplib

# ...

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

# ...
```

구성 설정

STP 라이브러리 set_config() 메소드를 사용하여 구성 설정을 수행합니다. 여기 예제로 다음 값을 설정합니다.

OpenFlow 스위치	항목	설정
dpid=0000000000000001	bridge.priority	0x8000
dpid=0000000000000002	bridge.priority	0x9000
dpid=0000000000000003	bridge.priority	0xa000

이 설정은 dpid = 0000000000000001의 OpenFlow 스위치의 브리지 ID가 항상 최소값이 루트 브리지에 선택되게 됩니다.

```
class SimpleSwitch13(app_manager.RyuApp):

# ...

def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self.stp = kwargs['stplib']

    # Sample of stplib config.
    # please refer to stplib.Stp.set_config() for details.
    config = {dpid_lib.str_to_dpid('0000000000000001'):
              {'bridge': {'priority': 0x8000}},
              dpid_lib.str_to_dpid('0000000000000002'):
              {'bridge': {'priority': 0x9000}},
              dpid_lib.str_to_dpid('0000000000000003'):
              {'bridge': {'priority': 0xa000}}}
    self.stp.set_config(config)
```

STP 이벤트 처리

「링크 어그리게이션」과 같이 STP 라이브러리의 통지가 이벤트를 수신하는 이벤트 처리기를 제공합니다.

STP 라이브러리에 정의 된 stplib.EventPacketIn 이벤트를 이용하여 BPDU 패킷을 제외한 패킷을 수신 할 수 있기 때문에, 「스위칭 허브」와 같은 패킷 핸드 연결합니다.

```
class SimpleSwitch13(app_manager.RyuApp):
    @set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # ...
```

네트워크 토플로지 변경 알림 이벤트 (stplib.EventTopologyChange)를 받아 학습 된 MAC 주소 및 등록 된 흐름 항목을 초기화합니다.

```
class SimpleSwitch13(app_manager.RyuApp):
    def delete_flow(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        for dst in self.mac_to_port[datapath.id].keys():
            match = parser.OFPMatch(eth_dst=dst)
            mod = parser.OFFlowMod(
                datapath, command=ofproto.OFPFC_DELETE,
                out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY,
                priority=1, match=match)
            datapath.send_msg(mod)

    # ...

    @set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
    def _topology_change_handler(self, ev):
        dp = ev.dp
        dpid_str = dpid_lib.dpid_to_str(dp.id)
        msg = 'Receive topology change event. Flush MAC table.'
        self.logger.debug("[dpid=%s] %s", dpid_str, msg)

        if dp.id in self.mac_to_port:
            self.delete_flow(dp)
            del self.mac_to_port[dp.id]
```

포트 상태 변경 알림 이벤트 (stplib.EventPortStateChange)를 받고 포트 상태 디버그 로그 출력을 실시하고 있습니다.

```
class SimpleSwitch13(app_manager.RyuApp):
    @set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
    def _port_state_change_handler(self, ev):
        dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
        of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                    stplib.PORT_STATE_BLOCK: 'BLOCK',
                    stplib.PORT_STATE_LISTEN: 'LISTEN',
                    stplib.PORT_STATE_LEARN: 'LEARN',
                    stplib.PORT_STATE_FORWARD: 'FORWARD'}
        self.logger.debug("[dpid=%s] [port=%d] state=%s",
                          dpid_str, ev.port_no, of_state[ev.port_state])
```

이상과 같이 스패닝 트리 기능을 제공하는 라이브러리와 라이브러리를 사용하는 응용 프로그램에서 스패닝 트리 기능을 가진 스위칭 허브 응용 프로그램을 실현하고 있습니다.

6.5 정리

이 장에서는 스패닝 트리 라이브러리 사용을 주제로 다음 항목 대해 설명했습니다.

- hub.Event을 이용한 이벤트 대기 처리의 실현 방법

- hub.Timeout를 이용한 타이머 제어 처리의 실현 방법

IGMP 스누핑

이 장에서는 Ryu 를 이용한 IGMP 스누핑 기능 을 구현하는 방법 을 설명 하고 있습니다.

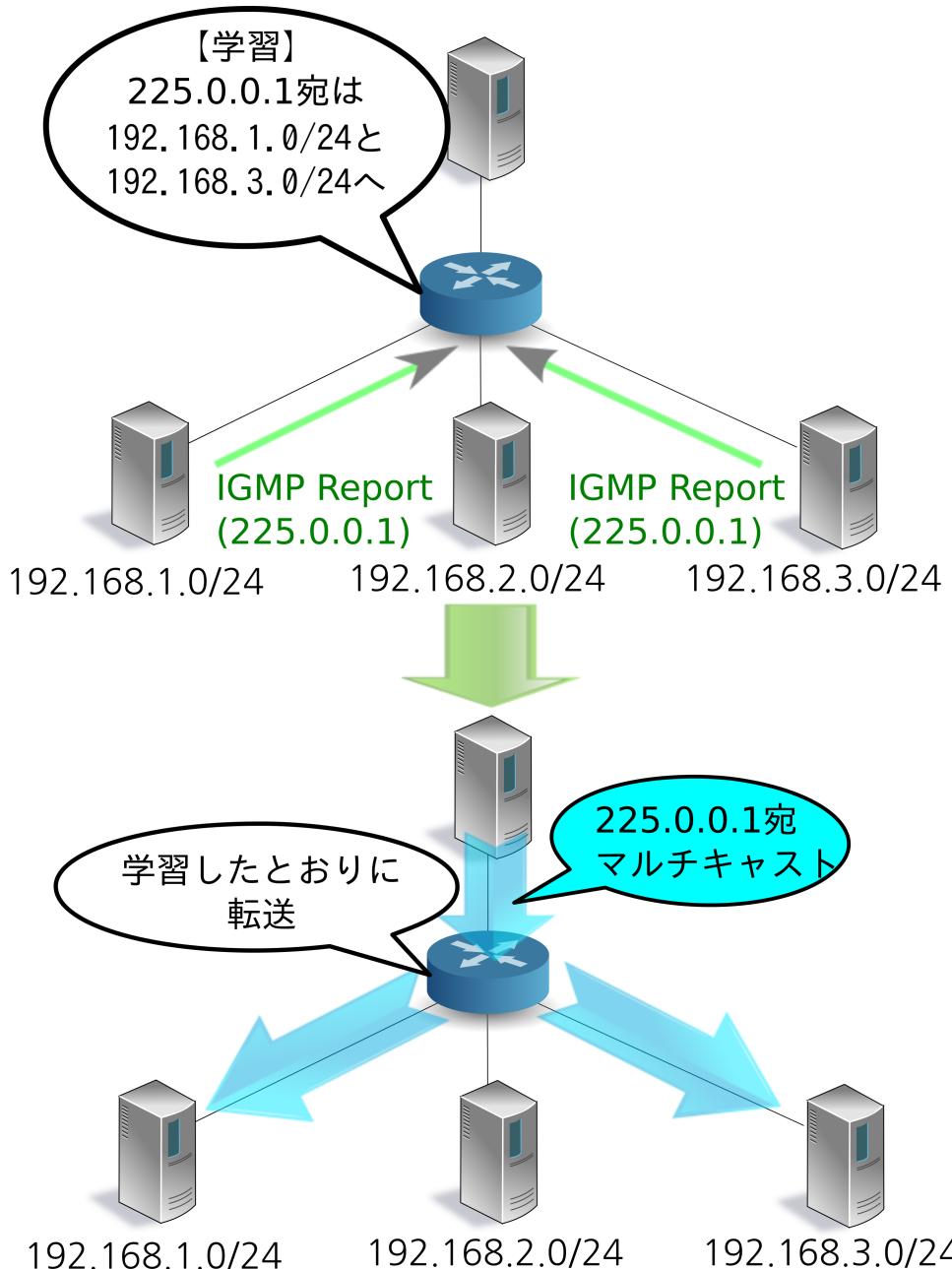
7.1 IGMP 스누핑

7.1.1 IGMP 내용

IGMP (Internet Group Management Protocol) 는 서브넷 간에 마루찌캬 스트 패킷 의 목적지를 관리하기위한 프로토콜입니다.

멀티 캐스트 라우터 는 라우터 가 연결된 모든 서브넷 에 대해 마루찌캬 스트 그룹 참여 호스트 가 있는지 여부를 주기적으로 요청 합니다 (IGMP Query Message) . 멀티 캐스트 그룹에 참여하는 호스트 가 어떤 사부넷토 에 존재 하는 경우 해당 호스트는 어느 멀티 캐스트 그룹에 참여하는 인지 멀티 캐스트 라우터에 보고합니다 (IGMP Report Message) . 멀티 캐스트 라우터 는 수신 한 보고가 어느 서브넷에서 보내진 것인지를 기억하고 ” 어떤 멀티 캐스트 그룹에 패킷을 어떻게 서브넷을 향해 전달할지 ” 를 결정합니다. 질문 맞게 대한 보고가 없거나 또는 특정 멀티 캐스트 그룹에서 탈퇴 메시지 (IGMP Leave Message) 를 호스트로부터 받은 경우 마루찌캬 스토루타는 해당 서브넷에 대한 모든 , 또는 지정된 멀티 캐스트 그루프 에게 패킷을 전달 하지 않습니다.

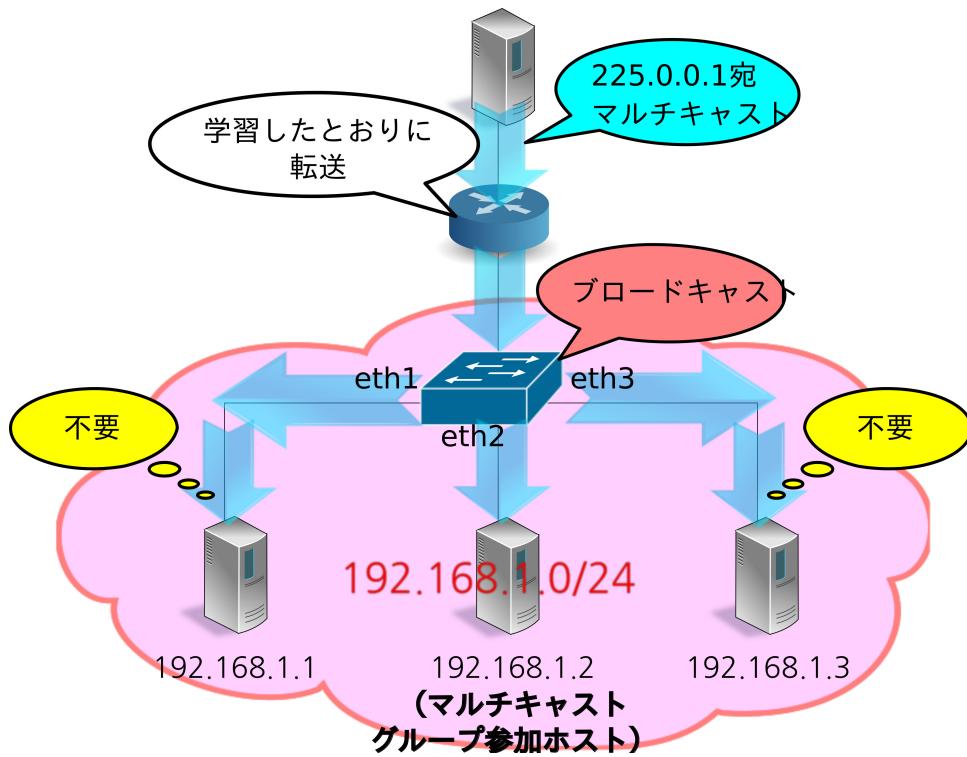
이 방식 은 멀티 캐스트 그룹 참여 호스트 가 없는 서브넷에 대해 라고 멀티 캐스트 패킷이 전송 되지 않으며 불필요한 트래픽을 감소 해야 할 수 있습니다.



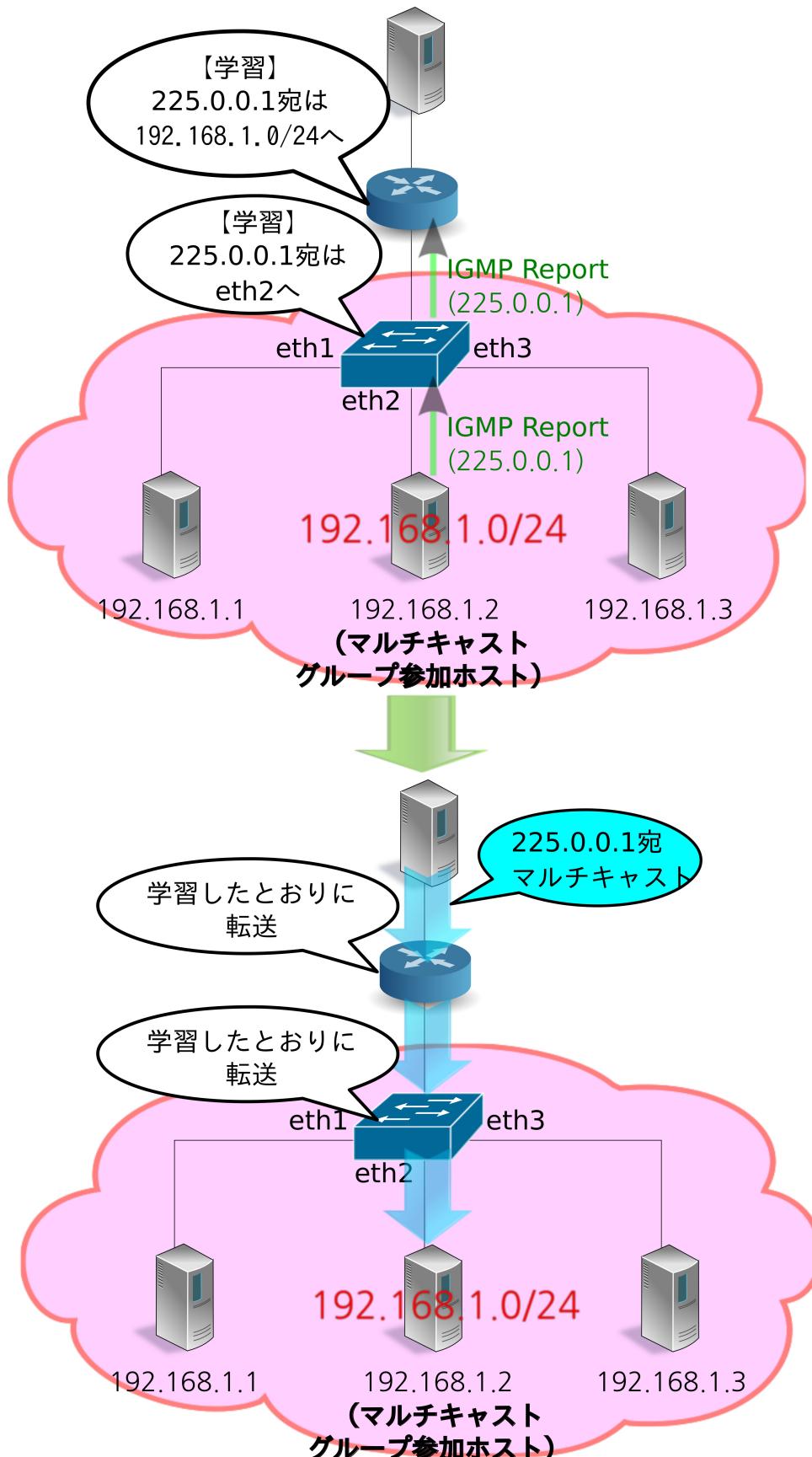
7.1.2 서브넷의 과제와 IGMP 스누핑 내용

IGMP를 사용하여 서브넷별로 불필요한 트래픽을 줄일 수 있습니다. 하지만, 서브넷 내에서 여전히 불필요한 트래픽이 발생할 수 있습니다.

멀티 캐스트 패킷의 목적지 MAC 주소는 특수한 값이기 때문에 L2 스위치의 MAC 주소 테이블에서 학습되는 것은 없고, 항상 방송 대상이 됩니다. 때문에, 예를 들어 있는 하나의 포트에만 멀티 캐스트 그룹 참여 호스트 연결 되고 있었다고 해도, L2 스위치는 수신 된 멀티 캐스트 패킷을 모든 포트로 전송 해 버립니다.



IGMP 스누핑은 멀티 캐스트 그룹 참여 호스트에서 멀티 캐스트 라우터에 전송되는 IGMP Report Message를 L2 스위치가 엿보기 (snoop) 것으로 마루찌캐스트 패킷의 대상 포트를 학습하는 수법입니다. 이 방법은 서브넷 트 내에서 멀티 캐스트 그룹 참여 호스트가 존재하지 않는 포트에 멀티 캐스트 패킷이 전송되는 것은 없으며 불필요한 트래픽을 줄일 수 있습니다.



IGMP 스누핑을 할 L2 스위치는 여러 호스트에서 동일한 멀티 캐스트 그 루프에 참가하고 있다는 IGMP Report Message를 수신해도 쿼리 기 1 드라이버 또는 IGMP Report Message를 전송하지 않습니다. 또한 호스트에서 IGMP Leave Message를 수신하고, 다른 동일한 멀티 캐스트 그룹에 참여하는 호스트가 존재하

는 동안는 쿼리 기에 IGMP Leave Message를 전송하지 않습니다. 이렇게하면 쿼리 기에는 아타 오리 단일 호스트와 IGMP 메시지를 교환하는 것처럼 보이게 할 수 있고, 또한 불필요한 IGMP 메시지의 전송을 억제 할 수 있습니다.

7.2 Ryu 응용 프로그램의 실행

IGMP 스누핑 기능을 OpenFlow를 이용하여 실현 한 Ryu의 IGMP 스누핑 앱 프로그램을 실행 해 봅니다.

Ryu 소스 트리에 포함되어있는 simple_switch_igmp.py는 OpenFlow 1.0 전용 응용 프로그램이기 때문에 여기에서는 새롭게 OpenFlow 1.3에 대응 한 simple_switch_igmp_13.py을 만들합니다. 이 프로그램은 「스위칭 허브」에 IGMP 스누핑 기능을 추가 한 응용 프로그램입니다. 또한이 프로그램은 dpid = 0000000000000001 스위치를 멀티 캐스트 라우터로 취급하고, 포트 2에 연결되어있는 호스트를 멀티 캐스트 서버로 처리하도록 설정되어 있습니다.

소스 이름: simple_switch_igmp_13.py

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import igmplib
from ryu.lib.dpid import str_to_dpid
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleSwitchIgmp13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'igmpplib': igmpplib.IgmpLib}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitchIgmp13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self._snoop = kwargs['igmpplib']
        self._snoop.set_querier_mode(
            dpid=str_to_dpid('0000000000000001'), server_port=2)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                         ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
```

```

inst = [parser.OFPIInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                      actions)]

mod = parser.OFPPFlowMod(datapath=datapath, priority=priority,
                         match=match, instructions=inst)
datapath.send_msg(mod)

@set_ev_cls(igmplib.EventPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ether.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                             in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)

@set_ev_cls(igmplib.EventMulticastGroupStateChanged,
            MAIN_DISPATCHER)
def _status_changed(self, ev):
    msg = {
        igmplib.MG_GROUP_ADDED: 'Multicast Group Added',
        igmplib.MG_MEMBER_CHANGED: 'Multicast Group Member Changed',
        igmplib.MG_GROUP_REMOVED: 'Multicast Group Removed',
    }
    self.logger.info("%s: [%s] querier:[%s] hosts:%s",
                    msg.get(ev.reason), ev.address, ev.src,
                    ev.dsts)

```

주석: 다음 예에서는 멀티 캐스트 패킷 송수신에 VLC (<http://www.videolan.org/vlc/>)를 사용합니다. VLC 설치, 및 스트

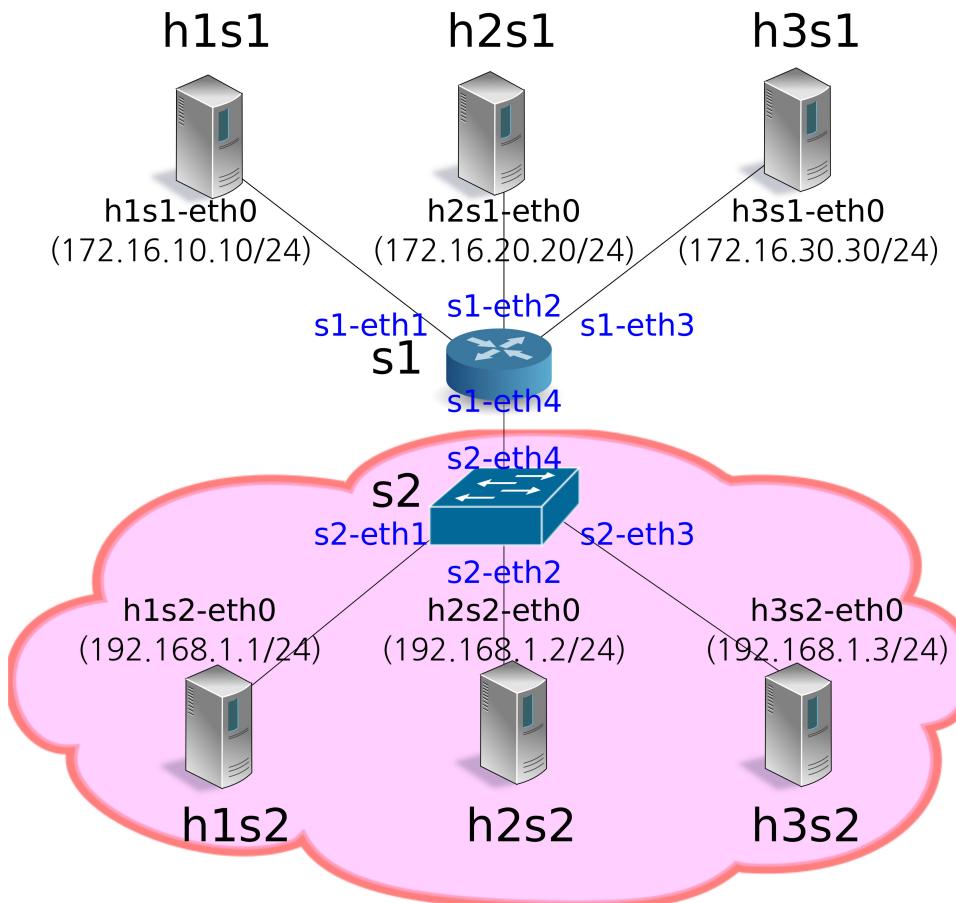
리밍 용 동영상을 구하는 관해서는 본고에서는 설명하지 않습니다.

7.2.1 실험 환경 구축

IGMP 스누핑 어플리케이션을 테스트하는 실험 환경을 구축합니다.

VM 이미지 사용을 위한 환경 설정 및 로그인 방법 등은 「[스위칭 허브](#)」을 참조하십시오.

먼저 Mininet를 이용하여 아래 그림과 같은 토폴로지를 만듭니다.



mn 명령의 매개 변수는 다음과 같습니다.

매개변수	값	설명
topo	linear,2,3	2 개의 스위치가 직렬로 연결되는 토폴로지 (각 스위치에 3 개의 호스트가 연결되는)
mac	없음	자동으로 호스트의 MAC 주소를 설정한다
switch	ovsk	Open vSwitch를 사용함
controller	remote	OpenFlow 컨트롤러는 외부의 것을 이용하기
x	없음	xterm을 시작

실행 예는 다음과 같습니다.

```
ryu@ryu-vm:~$ sudo mn --topo linear,2,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1s1 h1s2 h2s1 h2s2 h3s1 h3s2
*** Adding switches:
s1 s2
```

```
*** Adding links:  
(h1s1, s1) (h1s2, s2) (h2s1, s1) (h2s2, s2) (h3s1, s1) (h3s2, s2) (s1, s2)  
*** Configuring hosts  
h1s1 h1s2 h2s1 h2s2 h3s1 h3s2  
*** Running terms on localhost:10.0  
*** Starting controller  
*** Starting 2 switches  
s1 s2  
  
*** Starting CLI:  
mininet>
```

net 명령의 실행 결과는 다음과 같습니다.

```
mininet> net  
h1s1 h1s1-eth0:s1-eth1  
h1s2 h1s2-eth0:s2-eth1  
h2s1 h2s1-eth0:s1-eth2  
h2s2 h2s2-eth0:s2-eth2  
h3s1 h3s1-eth0:s1-eth3  
h3s2 h3s2-eth0:s2-eth3  
s1 lo: s1-eth1:h1s1-eth0 s1-eth2:h2s1-eth0 s1-eth3:h3s1-eth0 s1-eth4:s2-eth4  
s2 lo: s2-eth1:h1s2-eth0 s2-eth2:h2s2-eth0 s2-eth3:h3s2-eth0 s2-eth4:s1-eth4  
c0  
mininet>
```

7.2.2 IGMP 버전 설정

Ryu의 IGMP 스누핑 응용 프로그램은 IGMP v1/v2 만 지원합니다. 각 호스트가 IGMP v3을 사용하지 않도록 설정합니다. 이 명령 입력은 각 호스트의 xterm에서 실시해주세요.

host: h1s1:

```
root@ryu-vm:~# echo 2 > /proc/sys/net/ipv4/conf/h1s1-eth0/force_igmp_version
```

host: h1s2:

```
root@ryu-vm:~# echo 2 > /proc/sys/net/ipv4/conf/h1s2-eth0/force_igmp_version
```

host: h2s1:

```
root@ryu-vm:~# echo 2 > /proc/sys/net/ipv4/conf/h2s1-eth0/force_igmp_version
```

host: h2s2:

```
root@ryu-vm:~# echo 2 > /proc/sys/net/ipv4/conf/h2s2-eth0/force_igmp_version
```

host: h3s1:

```
root@ryu-vm:~# echo 2 > /proc/sys/net/ipv4/conf/h3s1-eth0/force_igmp_version
```

host: h3s2:

```
root@ryu-vm:~# echo 2 > /proc/sys/net/ipv4/conf/h3s2-eth0/force_igmp_version
```

7.2.3 IP 주소 설정

Mininet에 의해 자동으로 할당 된 IP 주소는 모든 호스트가 같은 서브 인터넷에 소속되어 있습니다. 다른 서브넷을 구축하기위한 각 호스트에서 IP 주소를 다시 할당합니다.

host: h1s1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1s1-eth0
root@ryu-vm:~# ip addr add 172.16.10.10/24 dev h1s1-eth0
```

host: h1s2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h1s2-eth0
root@ryu-vm:~# ip addr add 192.168.1.1/24 dev h1s2-eth0
```

host: h2s1:

```
root@ryu-vm:~# ip addr del 10.0.0.3/8 dev h2s1-eth0
root@ryu-vm:~# ip addr add 172.16.20.20/24 dev h2s1-eth0
```

host: h2s2:

```
root@ryu-vm:~# ip addr del 10.0.0.4/8 dev h2s2-eth0
root@ryu-vm:~# ip addr add 192.168.1.2/24 dev h2s2-eth0
```

host: h3s1:

```
root@ryu-vm:~# ip addr del 10.0.0.5/8 dev h3s1-eth0
root@ryu-vm:~# ip addr add 172.16.30.30/24 dev h3s1-eth0
```

host: h3s2:

```
root@ryu-vm:~# ip addr del 10.0.0.6/8 dev h3s2-eth0
root@ryu-vm:~# ip addr add 192.168.1.3/24 dev h3s2-eth0
```

7.2.4 기본 게이트웨이 설정

각 호스트에서 IGMP 패킷이 성공적으로 보낼 수 있도록 기본 게이트웨이를 설정 정합니다.

host: h1s1:

```
root@ryu-vm:~# ip route add default via 172.16.10.254
```

host: h1s2:

```
root@ryu-vm:~# ip route add default via 192.168.1.254
```

host: h2s1:

```
root@ryu-vm:~# ip route add default via 172.16.20.254
```

host: h2s2:

```
root@ryu-vm:~# ip route add default via 192.168.1.254
```

host: h3s1:

```
root@ryu-vm:~# ip route add default via 172.16.30.254
```

host: h3s2:

```
root@ryu-vm:~# ip route add default via 192.168.1.254
```

7.2.5 OpenFlow 버전 설정

사용하는 OpenFlow 버전을 1.3으로 설정합니다. 이 명령 입력 스위치 s1, s2의 xterm에서 실시해주세요.

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
switch: s2 (root):
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

7.2.6 스위칭 허브의 실행

준비가 그래서 처음에 만든 Ryu 응용 프로그램을 실행합니다. 이 명령 입력은 컨트롤러 c0의 xterm에서 실시해주세요.

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ./simple_switch_igmp_13.py
loading app ./simple_switch_igmp_13.py
loading app ryu.controller.ofp_handler
loading app ryu.controller.ofp_handler
instantiating app None of IgmpLib
creating context igmplib
instantiating app ./simple_switch_igmp_13.py of SimpleSwitchIgmp13
instantiating app ryu.controller.ofp_handler of OFPHandler
...
```

시작 후 바로 스위치 s1이 멀티 캐스트 라우터 (IGMP Query Message를 전송하기 때문에 쿼리 기라고도 함) 역할을 시작했다는 것을 나타내는 로그가 출력됩니다.

controller: c0 (root):

```
...
[querier] [INFO] started a querier.
...
```

쿼리 기는 60 초에 한 번 IGMP Query Message를 모든 포트로 전송하여 IGMP Report Message 이 돌아온 포트에 멀티 캐스트 서버에서 멀티 캐스트 패킷을 회전 전송 흐름 항목을 등록합니다.

동시에 쿼리 기 이외의 스위치에서 IGMP 패킷 스누핑이 시작됩니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000002 PORT=4 IGMP received. [QUERY]
...
```

위의 로그는 쿼리 기이다 스위치 s1에서 보낸 IGMP Query Message를 스 스위치 s2가 포트 4에서 수신 된 것을 나타냅니다. 스위치 s2는 받은 IGMP Query Message를 브로드 캐스트합니다.

주의: 스누핑의 준비가 있기 전에 쿼리 기에서 처음 IGMP Query Message 가 전송 될 수 있습니다. 이 경우 60 초 후에 보낸 다음 IGMP Query Message를 기다립니다.

7.2.7 멀티 캐스트 그룹 추가

이어 각 호스트를 멀티 캐스트 그룹에 참가합니다. VLC에서 특정 멀티 캐스트 트 주소로 보내는 스트림을 재생하려고했을 때, VLC는 IGMP Report Message를 전송 신합니다.

호스트 h1s2를 225.0.0.1 그룹에 가입시킴

우선 호스트 h1s2에서 멀티 캐스트 주소 「225.0.0.1」에게 스트림을 재생하도록 설정합니다. VLC는 호스트 h1s2에서 IGMP Report Message를 보냅니다.

host: h1s2:

```
root@ryu-vm:~# vlc-wrapper udp://@225.0.0.1
```

스위치 s2는 호스트 h1s2에서 IGMP Report Message를 포트 1로 수신하여 멀티 캐스트 주소 「225.0.0.1」을 받는 그룹이 포트 1 먼저 존재하는 것을 인식합니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000002 PORT=1 IGMP received. [REPORT]
Multicast Group Added: [225.0.0.1] querier:[4] hosts:[]
Multicast Group Member Changed: [225.0.0.1] querier:[4] hosts:[1]
[snoop] [INFO] SW=0000000000000002 PORT=1 IGMP received. [REPORT]
[snoop] [INFO] SW=0000000000000002 PORT=1 IGMP received. [REPORT]
...

```

위의 로그는 스위치 s2에 있어서

- IGMP Report Message를 포트 1에 받은 것
- 멀티 캐스트 주소 「225.0.0.1」을 수신하는 멀티 캐스트 그룹의 존재를 인식하는지 (스위치가 대기 포트 4 먼저 존재하는 것)
- 멀티 캐스트 주소 「225.0.0.1」을 수신하는 그룹의 참여 호스트 포트 1 끝에 있는지

을 나타냅니다. VLC는 시작할 때 IGMP Report Message를 3 회 보내 로그도 그렇게되어 있습니다.

이 후 쿼리 기는 60 초에 한 번 IGMP Query Message를 계속 보내는 메시지를 수신 멀티 캐스트 그룹 참여 호스트 h1s2 그때마다 IGMP Report Message를 전송 합니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000002 PORT=4 IGMP received. [QUERY]
[snoop] [INFO] SW=0000000000000002 PORT=1 IGMP received. [REPORT]
...

```

이 시점에서 쿼리 기 등록 된 흐름 항목을 확인하려고합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=827.211s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=2,nw_d...
cookie=0x0, duration=827.211s, table=0, n_packets=14, n_bytes=644, priority=65535, ip,in_port=4,nw_d...
cookie=0x0, duration=843.887s, table=0, n_packets=1, n_bytes=46, priority=0 actions=CONTROLLER:6
```

쿼리 기에

- 포트 2(멀티 캐스트 서버)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 4(스위치 s2)로 전송
- 포트 4(스위치 s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 「[스위칭 허브](#)」와 같은 Table-miss 흐름 항목

세 가지 흐름 항목이 등록되어 있습니다.

또한 스위치 s2에 등록되어 있는 흐름 항목도 확인 해 봅니다.

switch: s2 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=1463.549s, table=0, n_packets=26, n_bytes=1196, priority=65535, ip,in_port=1,nw_d...
cookie=0x0, duration=1463.548s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=4,nw_d...
cookie=0x0, duration=1480.221s, table=0, n_packets=26, n_bytes=1096, priority=0 actions=CONTROLLER:6
```

스위치 s2에

- 포트 1(호스트 h1s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 4(쿼리 기)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 1(포스 트 h1s2)로 전송

- 「스위칭 허브」와 같은 Table-miss 흐름 항목 세 가지 흐름 항목이 등록되어 있습니다.

호스트 h3s2를 225.0.0.1 그룹에 가입시킴

이어 호스트 h3s2에서도 멀티 캐스트 주소 「225.0.0.1」에게 스트림을 재생하도록 설정합니다. VLC는 호스트 h3s2에서 IGMP Report Message를 보냅니다.

host: h3s2:

```
root@ryu-vm:~# vlc-wrapper udp://@225.0.0.1
```

스위치 s2는 호스트 h3s2에서 IGMP Report Message를 포트 3에서 수신하고 마루찌카 스트 주소 「225.0.0.1」을 수신하는 그룹의 참여 호스트가 포트 1의 다른 포트 3 끝에도 존재 함을 인식합니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000000 PORT=3 IGMP received. [REPORT]
Multicast Group Member Changed: [225.0.0.1] querier:[4] hosts:[1, 3]
[snoop] [INFO] SW=0000000000000000 PORT=3 IGMP received. [REPORT]
[snoop] [INFO] SW=0000000000000000 PORT=3 IGMP received. [REPORT]
...

```

이 시점에서 쿼리 기 등록 된 흐름 항목을 확인하려고 합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=1854.016s, table=0, n_packets=0, n_bytes=0, priority=65535, ip, in_port=2, nw_
cookie=0x0, duration=1854.016s, table=0, n_packets=31, n_bytes=1426, priority=65535, ip, in_port=4
cookie=0x0, duration=1870.692s, table=0, n_packets=1, n_bytes=46, priority=0 actions=CONTROLLER:0
...
```

쿼리 기 등록 된 흐름 항목은 특히 변경은 없습니다.

또한 스위치 s2에 등록되어 있는 흐름 항목도 확인 해 봅니다.

switch: s2 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=1910.703s, table=0, n_packets=34, n_bytes=1564, priority=65535, ip, in_port=1
cookie=0x0, duration=162.606s, table=0, n_packets=5, n_bytes=230, priority=65535, ip, in_port=3, nw_
cookie=0x0, duration=162.606s, table=0, n_packets=0, n_bytes=0, priority=65535, ip, in_port=4, nw_d
cookie=0x0, duration=1927.375s, table=0, n_packets=35, n_bytes=1478, priority=0 actions=CONTROLLER:0
...
```

스위치 s2에

- 포트 1(호스트 h1s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 3(호스트 h3s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 4(쿼리 기)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 1(포스 트 h1s2) 및 포트 3(호스 트 h3s2)로 전송
- 「스위칭 허브」와 같은 Table-miss 흐름 항목

4 개의 흐름 항목이 등록되어 있습니다.

호스트 h2s2를 225.0.0.2 그룹에 가입시킴

그런 다음 호스트 h2s2는 다른 호스트와는 다른 멀티 캐스트 주소 「225.0.0.2」에게 스트림을 재생하도록 설정합니다. VLC는 호스트 h2s2에서 IGMP Report Message를 보냅니다.

host: h2s2:

```
root@ryu-vm:~# vlc-wrapper udp://@225.0.0.2
```

그런 다음 호스트 h2s2는 다른 호스트와는 다른 멀티 캐스트 주소 「225.0.0.2」에게 스트림을 재생하도록 설정합니다. VLC는 호스트 h2s2에서 IGMP Report Message를 보냅니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000002 PORT=2 IGMP received. [REPORT]
Multicast Group Added: [225.0.0.2] querier:[4] hosts:[]
Multicast Group Member Changed: [225.0.0.2] querier:[4] hosts:[2]
[snoop] [INFO] SW=0000000000000002 PORT=2 IGMP received. [REPORT]
[snoop] [INFO] SW=0000000000000002 PORT=2 IGMP received. [REPORT]
...

```

이 시점에서 쿼리 기 등록된 흐름 항목을 확인하려고 합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=2289.168s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=2,nw_
cookie=0x0, duration=108.374s, table=0, n_packets=2, n_bytes=92, priority=65535, ip,in_port=4,nw_
cookie=0x0, duration=108.375s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=2,nw_d_
cookie=0x0, duration=2289.168s, table=0, n_packets=38, n_bytes=1748, priority=65535, ip,in_port=4,nw_
cookie=0x0, duration=2305.844s, table=0, n_packets=2, n_bytes=92, priority=0 actions=CONTROLLER:0
```

쿼리 기에

- 포트 2(멀티 캐스트 서버)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 4(스위치 s2)로 전송
- 포트 4(스위치 s2)에서 225.0.0.2에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 2(멀티 캐스트 서버)에서 225.0.0.2에게 패킷을 수신 한 경우에는 포트 4(스위치 s2)로 전송
- 포트 4(스위치 s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 「스위칭 허브」와 같은 Table-miss 흐름 항목

5 개의 흐름 항목이 등록되어 있습니다.

또한 스위치 s2에 등록되어 있는 흐름 항목도 확인 해 봅니다.

switch: s2 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=2379.973s, table=0, n_packets=41, n_bytes=1886, priority=65535, ip,in_port=1
cookie=0x0, duration=199.178s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=4,nw_d_
cookie=0x0, duration=631.876s, table=0, n_packets=12, n_bytes=552, priority=65535, ip,in_port=3,nw_d_
cookie=0x0, duration=199.178s, table=0, n_packets=5, n_bytes=230, priority=65535, ip,in_port=2,nw_d_
cookie=0x0, duration=631.876s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=4,nw_d_
cookie=0x0, duration=2396.645s, table=0, n_packets=43, n_bytes=1818, priority=0 actions=CONTROLLER:0
```

스위치 s2에

- 포트 1(호스트 h1s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 4(쿼리 기)에서 225.0.0.2에게 패킷을 수신 한 경우에는 포트 2(호스 트 h2s2)로 전송
- 포트 3(호스트 h3s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 2(호스트 h2s2)에서 225.0.0.2에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 4(쿼리 기)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 1(포스 트 h1s2) 및 포트 3(호스 트 h3s2)로 전송
- 「스위칭 허브」와 같은 Table-miss 흐름 항목

6 개의 흐름 항목이 등록되어 있습니다.

호스트 h3s1를 225.0.0.1 그룹에 가입시킴

또한 호스트 h3s1에서도 멀티 캐스트 주소 「225.0.0.1」에게 스트림을 재생하도록 설정합니다. VLC는 호스트 h3s1에서 IGMP Report Message를 보냅니다.

host: h3s1:

```
root@ryu-vm:~# vlc-wrapper udp://@225.0.0.1
```

호스트 h3s1 스위치 s2와 연결되어 있지 않습니다. 따라서 IGMP 스누핑 기능의 대상이 아니라 쿼리 기사의 일반적 IGMP 패킷의 교환을 실시합니다.

이 시점에서 쿼리 기 등록된 흐름 항목을 확인하려고 합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=12.85s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=2,nw_dst=225.0.0.1
cookie=0x0, duration=626.33s, table=0, n_packets=10, n_bytes=460, priority=65535, ip,in_port=4,nw_dst=225.0.0.1
cookie=0x0, duration=12.85s, table=0, n_packets=1, n_bytes=46, priority=65535, ip,in_port=3,nw_dst=225.0.0.1
cookie=0x0, duration=626.331s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=2,nw_dst=225.0.0.1
cookie=0x0, duration=2807.124s, table=0, n_packets=46, n_bytes=2116, priority=65535, ip,in_port=4,nw_dst=225.0.0.1
cookie=0x0, duration=2823.8s, table=0, n_packets=3, n_bytes=138, priority=0 actions=CONTROLLER:65535
```

쿼리 기에

- **포트 2(멀티 캐스트 서버)**에서 **225.0.0.1**에게 패킷을 수신 한 경우에는 포트 3(h3s1) 및 포트 4(스위치 s2)로 전송
- **포트 4(스위치 s2)**에서 **225.0.0.2**에게 패킷을 수신 한 경우에는 **Packet-In** 하는
- 포트 3(h3s1)에서 225.0.0.1에게 패킷을 수신 한 경우에는 **Packet-In**하기
- **포트 2(멀티 캐스트 서버)**에서 **225.0.0.2**에게 패킷을 수신 한 경우에는 포트 4(스위치 s2)로 전송
- **포트 4(스위치 s2)**에서 **225.0.0.1**에게 패킷을 수신 한 경우에는 **Packet-In** 하는
- 「**스위칭 허브**」와 같은 Table-miss 흐름 항목

6 개의 흐름 항목이 등록되어 있습니다.

7.2.8 스트리밍 시작

멀티 캐스트 서버 인 호스트 h2s1에서 스트리밍을 시작합니다. 멀티 캐스트 주소에는 「225.0.0.1」을 사용하기로 합니다.

host: h2s1:

```
root@ryu-vm:~# vlc-wrapper sample.mov --sout udp:225.0.0.1 --loop
```

그려자 「225.0.0.1」 멀티 캐스트 그룹에 참여하는 h1s2, h3s2, h1s3 각 호스트에서 실행하는 VLC 멀티 캐스트 서버에서 제공하고 있는 동영상이 재생 됩니다. 「225.0.0.2」에 참여하는 h2s2에서 동영상이 재생되지 않습니다.

7.2.9 멀티 캐스트 그룹 삭제

이어 각 호스트를 멀티 캐스트 그룹에서 이탈시킵니다. 스트림 재생중인 VLC 을 종료했을 때 VLC는 IGMP Leave Message를 보냅니다.

호스트 h1s2를 225.0.0.1 그룹에서 이탈시킴

호스트 h1s2에서 실행중인 VLC를 Ctrl+C 등으로 종료합니다. 스위치 s2는 호스트 h1s2 또는 라 IGMP Leave Message를 포트 1로 수신하여 멀티 캐스트 주소 「225.0.0.1」를 수신하는 그룹의 참여 호스트가 포트 1의 끝에 존재하지 않는 것을 인식합니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000002 PORT=1 IGMP received. [LEAVE]
Multicast Group Member Changed: [225.0.0.1] querier:[4] hosts:[3]
...
```

위의 로그는 스위치 s2에 있어서

- 포트 1에서 IGMP Leave Message를 받은 것
- 멀티 캐스트 주소 「225.0.0.1」을 수신하는 그룹의 참여 호스트 포트 3 끝에 있는지

을 나타냅니다. IGMP Leave Message 수신 전까지는 멀티 캐스트 주소 「225.0.0.1」을 수신하는 그룹의 참여 호스트는 포트 1과 3의 끝에 존재하면 인식하고 있었습니다만, IGMP Leave Message 수신시 포트 1이 대상 외가되고 지금입니다.

이 시점에서 쿼리 기 등록 된 흐름 항목을 확인하려고 합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=1565.13s, table=0, n_packets=1047062, n_bytes=1421910196, priority=65535, ip, nw_
cookie=0x0, duration=2178.61s, table=0, n_packets=36, n_bytes=1656, priority=65535, ip, in_port=4, nw_
cookie=0x0, duration=1565.13s, table=0, n_packets=27, n_bytes=1242, priority=65535, ip, in_port=3, nw_
cookie=0x0, duration=2178.611s, table=0, n_packets=0, n_bytes=0, priority=65535, ip, in_port=2, nw_
cookie=0x0, duration=4359.404s, table=0, n_packets=72, n_bytes=3312, priority=65535, ip, in_port=4, nw_
cookie=0x0, duration=4376.08s, table=0, n_packets=3, n_bytes=138, priority=0 actions=CONTROLLER:
```

쿼리 기 등록 된 흐름 항목은 특히 변경은 없습니다.

또한 스위치 s2에 등록되어 있는 흐름 항목도 확인 해 봅니다.

switch: s2 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=2228.528s, table=0, n_packets=0, n_bytes=0, priority=65535, ip, in_port=4, nw_
cookie=0x0, duration=2661.226s, table=0, n_packets=46, n_bytes=2116, priority=65535, ip, in_port=3, nw_
cookie=0x0, duration=2228.528s, table=0, n_packets=39, n_bytes=1794, priority=65535, ip, in_port=2, nw_
cookie=0x0, duration=548.063s, table=0, n_packets=486571, n_bytes=660763418, priority=65535, ip, in_port=1, nw_
cookie=0x0, duration=4425.995s, table=0, n_packets=78, n_bytes=3292, priority=0 actions=CONTROLLER:
```

스위치 s2에

- 포트 4(쿼리 기)에서 225.0.0.2에게 패킷을 수신 한 경우에는 포트 2(호스트 h2s2)로 전송
- 포트 3(호스트 h3s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 2(호스트 h2s2)에서 225.0.0.2에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 4(쿼리 기)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 3(호스트 h3s2)로 전송
- 「스위칭 허브」와 같은 Table-miss 흐름 항목

5 개의 흐름 항목이 등록되어 있습니다. 방금 전까지 비해

- 포트 1(호스트 h1s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는

흐름 항목이 삭제되고, 또한 쿼리 기에서 225.0.0.1에게 패킷 전송 끝에서 포트 1(호스트 h1s2)가 제외 된 것을 알 수 있습니다.

호스트 h3s2를 225.0.0.1 그룹에서 이탈시킴

그런 다음 호스트 h3s2에서 실행중인 VLC를 Ctrl + C 등으로 종료합니다. 스위치 s2는 호스트 h3s2로부터 IGMP Leave Message 포트 3에서 수신 멀티 캐스트 주소 「225.0.0.1」을 수신하는 그룹의 참여 호스트 포트 3 먼저 존재하지 않는 것을 인식합니다.

controller: c0 (root):

```
...
[snoop] [INFO] SW=0000000000000002 PORT=3 IGMP received. [LEAVE]
Multicast Group Removed: [225.0.0.1] querier:[4] hosts:[]
...
```

위의 로그는 스위치 s2에 있어서

- 포트 3에서 IGMP Leave Message를 받은 것
- 멀티 캐스트 주소 「225.0.0.1」을 수신하는 그룹의 참여 호스트가 모든 존재하지 않는 것을 나타냅니다.

이 시점에서 쿼리 기 등록 된 흐름 항목을 확인하려고 합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=89.891s, table=0, n_packets=79023, n_bytes=107313234, priority=65535, ip,in_port=1
cookie=0x0, duration=3823.61s, table=0, n_packets=64, n_bytes=2944, priority=65535, ip,in_port=4
cookie=0x0, duration=3210.139s, table=0, n_packets=55, n_bytes=2530, priority=65535, ip,in_port=3
cookie=0x0, duration=3823.467s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=2,nw_src=192.168.1.1
cookie=0x0, duration=6021.089s, table=0, n_packets=4, n_bytes=184, priority=0 actions=CONTROLLER
```

쿼리 기에

- 포트 2(멀티 캐스트 서버)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 3(h3s1)로 전송
- 포트 4(스위치 s2)에서 225.0.0.2에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 3(h3s1)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In하기
- 포트 2(멀티 캐스트 서버)에서 225.0.0.2에게 패킷을 수신 한 경우에는 포트 4(스위치 s2)로 전송
- 「[스위칭 허브](#)」와 같은 Table-miss 흐름 항목

5 개의 흐름 항목이 등록되어 있습니다. 방금 전까지 비해

- 포트 4(스위치 s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는

흐름 항목이 삭제되고, 또한 멀티 캐스트 서버에서 225.0.0.1에게 패킷의 대상에서 포트 4(스위치 s2)가 제외 된 것을 알 수 있습니다.

또한 스위치 s2에 등록되어 있는 흐름 항목도 확인 해 봅니다.

switch: s2 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=4704.052s, table=0, n_packets=0, n_bytes=0, priority=65535, ip,in_port=4,nw_src=192.168.1.1
cookie=0x0, duration=4704.053s, table=0, n_packets=53, n_bytes=2438, priority=65535, ip,in_port=2
cookie=0x0, duration=6750.068s, table=0, n_packets=115, n_bytes=29870, priority=0 actions=CONTROLLER
```

스위치 s2에

- 포트 4(쿼리 기)에서 225.0.0.2에게 패킷을 수신 한 경우에는 포트 2(호스트 h2s2)로 전송
- 포트 2(호스트 h2s2)에서 225.0.0.2에게 패킷을 수신 한 경우에는 Packet-In 하는
- 「[스위칭 허브](#)」와 같은 Table-miss 흐름 항목

세 가지 흐름 항목이 등록되어 있습니다. 방금 전까지 비해

- 포트 3(호스트 h3s2)에서 225.0.0.1에게 패킷을 수신 한 경우에는 Packet-In 하는
- 포트 4(쿼리 기)에서 225.0.0.1에게 패킷을 수신 한 경우에는 포트 3(포스트 h3s2)로 전송 흐름 항목이 삭제 된 것을 알 수 있습니다.

7.3 Ryu에 따르면 IGMP 스누핑 기능의 구현

OpenFlow를 사용하여 어떻게 IGMP 스누핑 기능을 수행하고 있는지를 살펴 보겠습니다.

IGMP 스누핑 「IGMP 패킷을 들여다 보는」라는 동작은 OpenFlow의 Packet-In 메시지를 사용하여 구현하고 있습니다. Packet-In 메시지에서 컨트롤러로 전송 된 IGMP 패킷의 내용을

- 어떤 그룹에 대한 IGMP 메시지인지
- IGMP Report Message인지 IGMP Leave Message인가
- 스위치의 모든 포트에서 수신하여 IGMP 메시지인지

라는 관점에서 분석하고 그에 따라 처리합니다.

프로그램은 멀티 캐스트 주소와 멀티 캐스트 주소로 보내는 패킷을 어느 포트로 전송하거나 대응표를 유지합니다.

IGMP Report Message를 수신했을 때, 그것이 해당 테이블에 존재하지 않는 포트로부터의 것 있으면 해당 멀티 캐스트 주소로 보내는 패킷을 해당 포트로 전송하는 후로엔 트리 등록합니다.

IGMP Leave Message를 수신했을 때, 그것이 대응표에 있는 포트에서의 것이 든 경우 확인 용 IGMP Query Message를 보내고 응답이 없으면 해당 멀티 캐스트 주소 레스에게 패킷을 해당 포트로 전송하는 흐름 항목을 삭제하십시오.

IGMP Leave Message를 보내지 않고 멀티 캐스트 그룹 참여 호스트가 결핍되었다 경우를 고려하여 쿼리 기에서의 IGMP Query Message를 전송 할 때마다 각 포트에서 IGMP Report Message가 돌아왔다 여부를 확인합니다. IGMP Report Message를 회신하지 않았다 포트의 끝에는 멀티 캐스트 그룹 참여 호스트가 존재하지 않는 것과 간주 멀티 캐스트 패킷을 해당 포트로 전송하는 흐름 항목을 삭제하십시오.

멀티 캐스트 그룹에 대한 IGMP Report Message를 여러 포트에서 수신 경우 프로그램은 첫 번째 메시지만 쿼리 기로 전송합니다. 이것은 불필요한 같은 IGMP Report Message를 쿼리 기에 전송하는 것을 억제합니다.

멀티 캐스트 그룹에 대한 IGMP Leave Message를 수신 한 경우 해당 그룹 루프에 참가하는 호스트가 다른 포트의 끝에 존재하는 경우 프로그램은 IGMP Leave Message를 쿼리 기에 전송하지 않습니다. 그 그룹에 참여하는 호스트가 하나도 없어 졌을 때, 프로그램은 IGMP Leave Message를 쿼리 기로 전송 있습니다. 이렇게하면 불필요한 IGMP Leave Message를 쿼리 기에 전송하는 것을 억제합니다입니다.

또한 쿼리 기이다 멀티 캐스트 라우터가 존재하지 않는 네트워크에서도 IGMP 스누핑 기능이 작동 할 수 있도록 의사 쿼리 기 기능도 구현하기로 합니다.

이상의 내용을 IGMP 스누핑 기능을 포괄적으로 구현하는 IGMP 스누핑 라이브러리 하면 라이브러리를 사용하는 응용 프로그램에 나누어 구현합니다.

IGMP 스누핑 라이브러리

- 스누핑 기능
 - IGMP Query Message를 받으면 보유하고 응답 여부의 정보를 초기화하고 IGMP Report Message를 기다리는
 - IGMP Report Message를 수신하면 해당 테이블을 갱신하고 필요하다면 흐름 항목 등록을 실시 한다. 또한 필요한 경우 쿼리 기 메시지를 전송하는
 - IGMP Leave Message를 받으면 확인에 대한 IGMP Query Message를 보내고 응답이 없으면 흐름 항목의 삭제를 실시한다. 또한 필요한 경우 쿼리 기 메시지를 전송하는
 - 보유하고 있는 해당 테이블이 업데이트되었을 때 그 사실을 응용 프로그램에 알리기 위해 이벤트를 보내는

- 의사 쿼리 기 기능
 - 정기적으로 IGMP Query Message를 플러딩, IGMP Report Message를 기다리는
 - IGMP Report Message를 받으면 흐름 항목의 등록을 할
 - IGMP Leave Message를 받으면 흐름 항목을 삭제할

응용 프로그램

- IGMP 스누핑 라이브러리의 통지를 받은 로그를 출력하는
- IGMP 패킷 이외의 패킷은 종래대로 학습 · 전송

IGMP 스누핑 라이브러리 및 응용 프로그램의 소스 코드, Ryu 소스 트리에 있습니다.

`ryu/lib/igmplib.py`

`ryu/app/simple_switch_igmp.py`

주석: `simple_switch_igmp.py`는 OpenFlow 1.0 전용 응용 프로그램이기 때문에 이 장에서는 「Ryu 응용 프로그램의 실행」으로 보여 주었다. OpenFlow 1.3에 대응 한 `simple_switch_igmp_13.py` 기반으로 응용 프로그램 자세한 내용을 설명합니다.

7.3.1 IGMP 스누핑 라이브러리 구현

다음 절에서는 위의 기능이 IGMP 스누핑 라이브러리에서 어떻게 구현 있는지를 살펴 보겠습니다. 또한 인용 된 소스는 발췌입니다. 전체 그림은 실제 소스를 참조하십시오.

스누핑 클래스와 의사 크 지역 클래스

라이브러리의 초기화 때 스누핑 클래스와 의사 크 지역 클래스를 인스턴스화 합니다.

```
def __init__(self):
    """Initialization."""
    super(IgmpLib, self).__init__()
    self.name = 'igmplib'
    self._querier = IgmpQuerier()
    self._snooper = IgmpSnooper(self.send_event_to_observers)
```

의사 크 지역 인스턴스에 대한 쿼리 기 역할을 하는 스위치의 설정과 멀티캐스트 서버 연결되는 포트 설정은 라이브러리의 메서드가 사용됩니다.

```
def set_querier_mode(self, dpid, server_port):
    """Set a datapath id and server port number to the instance
    of IgmpQuerier.

    =====
    Attribute   Description
    =====
    dpid        the datapath id that will operate as a querier.
    server_port the port number linked to the multicasting server.
    =====
    """
    self._querier.set_querier_mode(dpid, server_port)
```

의사 크 지역 인스턴스 스위치와 포트 번호가 지정된 경우 지정된 스위치가 응용 프로그램과 연결했을 때 의사 쿼리 기 처리를 시작합니다.

```
@set_ev_cls(ofp_event.EventOFPSStateChange,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def state_change_handler(self, evt):
    """StateChange event handler."""
    datapath = evt.datapath
```

```

assert datapath is not None
if datapath.id == self._querier.dpid:
    if evt.state == MAIN_DISPATCHER:
        self._querier.start_loop(datapath)
    elif evt.state == DEAD_DISPATCHER:
        self._querier.stop_loop()

```

Packet-In 처리

「 링크 어그리게이션 」뿐만 아니라 IGMP 패킷은 모든 IGMP 스누페 징 라이브러리에서 처리합니다. IGMP 패킷을 수신한 스위치가 의사 쿼리 아인스 옷장에 설정한 스위치 인 경우에는 의사 크 지역 인스턴스에 다른 장소 경우는 스누핑 인스턴스에 각각 처리를 맡기고 있습니다.

```

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, evt):
    """PacketIn event handler. when the received packet was IGMP,
    proceed it. otherwise, send a event."""
    msg = evt.msg
    dpid = msg.datapath.id

    req_pkt = packet.Packet(msg.data)
    req_igmp = req_pkt.get_protocol(igmp.igmp)
    if req_igmp:
        if self._querier.dpid == dpid:
            self._querier.packet_in_handler(req_igmp, msg)
        else:
            self._snooper.packet_in_handler(req_pkt, req_igmp, msg)
    else:
        self.send_event_to_observers(EventPacketIn(msg))

```

스누핑 인스턴스의 Packet-In 처리에서는 수신된 IGMP 패킷의 종류에 따라 라고 처리합니다.

```

def packet_in_handler(self, req_pkt, req_igmp, msg):
    #
    if igmp.IGMP_TYPE_QUERY == req_igmp.msgtype:
        self.logger.info(log + "[QUERY]")
        (req_ip4, ) = req_pkt.get_protocols(ipv4.ipv4)
        (req_eth, ) = req_pkt.get_protocols(ethernet.ethernet)
        self._do_query(req_igmp, req_ip4, req_eth, in_port, msg)
    elif (igmp.IGMP_TYPE_REPORT_V1 == req_igmp.msgtype or
          igmp.IGMP_TYPE_REPORT_V2 == req_igmp.msgtype):
        self.logger.info(log + "[REPORT]")
        self._do_report(req_igmp, in_port, msg)
    elif igmp.IGMP_TYPE_LEAVE == req_igmp.msgtype:
        self.logger.info(log + "[LEAVE]")
        self._do_leave(req_igmp, in_port, msg)
    #

```

의사 크 지역 인스턴스의 Packet-In 처리로 수신된 IGMP 패킷의 유형에 따라 글자 처리합니다.

```

def packet_in_handler(self, req_igmp, msg):
    #
    if (igmp.IGMP_TYPE_REPORT_V1 == req_igmp.msgtype or
        igmp.IGMP_TYPE_REPORT_V2 == req_igmp.msgtype):
        self._do_report(req_igmp, in_port, msg)
    elif igmp.IGMP_TYPE_LEAVE == req_igmp.msgtype:
        self._do_leave(req_igmp, in_port, msg)

```

스누핑 인스턴스에서 IGMP Query Message 처리

스누핑 인스턴스는 스위치 당 「쿼리 기와 연결된 포트」 「쿼리 기의 IP 주소」 「쿼리 기의 MAC 주소」 를 유지하는 영역이 있습니다. 또는 각 스위치에 대해 「알려진 멀티 캐스트 그룹」 「해당 멀티 캐스트 그룹에 참여하는 호스트가 연결되어 있는 포트 번호」 「메시지 수신 여부」 를 유지할 공간이 마련되어 있습니다.

스누핑 인스턴스는 IGMP Query Message 수신 시 메시지를 보내왔다 한 쿼리 기의 정보를 유지합니다.

또한 각 스위치의 메시지 수신 여부를 초기화 받은 IGMP Query Message 흥수 후 멀티 캐스트 그룹 참여 호스트에서 IGMP Report Message 수신 타임 아웃 처리를 실시합니다.

```
def _do_query(self, query, iph, eth, in_port, msg):
    # ...

    # learn the querier.
    self._to_querier[dpid] = {
        'port': in_port,
        'ip': iph.src,
        'mac': eth.src
    }

    # ...
    if '0.0.0.0' == query.address:
        # general query. reset all reply status.
        for group in self._to_hosts[dpid].values():
            group['replied'] = False
            group['leave'] = None
    # ...

    actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
    self._do_packet_out(
        datapath, msg.data, in_port, actions)

    # wait for REPORT messages.
    hub.spawn(self._do_timeout_for_query, timeout, datapath)
```

스누핑 인스턴스에서 IGMP Report Message 처리

스누핑 인스턴스는 멀티 캐스트 그룹 참여 호스트에서 IGMP Report Message를 수신했을 때 해당 멀티 캐스트 주소가 불명한 것이며 레바 멀티 캐스트 그룹 추가 이벤트를 전송하고 보존 영역을 업데이트합니다.

또한 해당 멀티 캐스트 그룹 IGMP Report Message를 포트에 처음 받은 경우에는 보존 영역을 업데이트하고 해당 멀티 캐스트 패킷의 대상으로 라고 그 포트를 추가 한 흐름 항목을 등록하고 멀티 캐스트 그룹 변경 이벤트 트를 보냅니다.

해당 멀티 캐스트 그룹의 IGMP Report Message를 아직 쿼리 기로 전송 있지 않하여야 전송합니다.

```
def _do_report(self, report, in_port, msg):
    # ...

    if not self._to_hosts[dpid].get(report.address):
        self._send_event(
            EventMulticastGroupStateChanged(
                MG_GROUP_ADDED, report.address, outport, []))
        self._to_hosts[dpid].setdefault(
            report.address,
            {'replied': False, 'leave': None, 'ports': {}})

    # ...
    if not self._to_hosts[dpid][report.address]['ports'].get(
        in_port):
        self._to_hosts[dpid][report.address]['ports'][in_port] = {'out': False, 'in': False}
        self._set_flow_entry(
```

```

        datapath,
        [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, size)],
        in_port, report.address)

if not self._to_hosts[dpid][report.address]['ports'][in_port]['out']:
    self._to_hosts[dpid][report.address]['ports'][in_port]['out'] = True

# ...
if not self._to_hosts[dpid][report.address]['ports'][in_port]['in']:
    actions = []
    ports = []
    for port in self._to_hosts[dpid][report.address]['ports']:
        actions.append(parser.OFPActionOutput(port))
        ports.append(port)
    self._send_event(
        EventMulticastGroupStateChanged(
            MG_MEMBER_CHANGED, report.address, outport, ports))
    self._set_flow_entry(
        datapath, actions, outport, report.address)
    self._to_hosts[dpid][report.address]['ports'][in_port]['in'] = True

# send a REPORT message to the querier if this message arrived
# first after a QUERY message was sent.
if not self._to_hosts[dpid][report.address]['replied']:
    actions = [parser.OFPActionOutput(outport, size)]
    self._do_packet_out(datapath, msg.data, in_port, actions)
    self._to_hosts[dpid][report.address]['replied'] = True

```

스누핑 인스턴스에서 IGMP Report Message 수신 타임 아웃 처리

IGMP Query Message 처리 후 일정 시간이 지나면 IGMP Report Message 수신 타임 아웃 처리 관리를 시작합니다. 멀티 캐스트 그룹 참여 호스트가 존재한다면 일반적 시간 초과가 발생하기 전에 IGMP Report Message를 보내 오기 위해, IGMP Report Message 처리에서 보존 영역의 업데이트가 이루어집니다.

일정 시간 경과 한 후에도 여전히 특정 멀티 캐스트 그룹에 대한 IGMP Report Message를 수신하지 못하면 해당 멀티 캐스트 그룹에 참가 호스트가 없어진 것으로 간주 멀티 캐스트 그룹 삭제 이벤트를 보내고, 흐름 항목의 삭제, 보존 영역의 업데이트를 수행합니다.

```

def _do_timeout_for_query(self, timeout, datapath):
    # ...
    hub.sleep(timeout)
    # ...

    remove_dsts = []
    for dst in self._to_hosts[dpid]:
        if not self._to_hosts[dpid][dst]['replied']:
            # if no REPORT message sent from any members of
            # the group, remove flow entries about the group and
            # send a LEAVE message if exists.
            self._remove_multicast_group(datapath, outport, dst)
            remove_dsts.append(dst)

    for dst in remove_dsts:
        del self._to_hosts[dpid][dst]

def _remove_multicast_group(self, datapath, outport, dst):
    # ...

```

```

self._send_event(
    EventMulticastGroupStateChanged(
        MG_GROUP_REMOVED, dst, outport, []))
self._del_flow_entry(datapath, outport, dst)
for port in self._to_hosts[dpid][dst]['ports']:
    self._del_flow_entry(datapath, port, dst)
#...

```

스누핑 인스턴스에서 IGMP Leave Message 처리

스누핑 인스턴스는 멀티 캐스트 그룹 참여 호스트에서 IGMP Leave Message를 수신했을 때, 보존 영역에 받은 메시지를 저장 한 후 확인에 대한 IGMP Query Message를 수신 한 포트로 전송하고 멀티 캐스트 그루프 참여 호스트에서 IGMP Report Message (Leave 응답) 수신 타임 아웃 처리를 행 있습니다.

```

def _do_leave(self, leave, in_port, msg):
    # ...

    self._to_hosts.setdefault(dpid, {})
    self._to_hosts[dpid].setdefault(
        leave.address,
        {'replied': False, 'leave': None, 'ports': {}})
    self._to_hosts[dpid][leave.address]['leave'] = msg
    self._to_hosts[dpid][leave.address]['ports'][in_port] = {
        'out': False, 'in': False}

    # ...
    # send a specific query to the host that sent this message.
    actions = [parser.OFPActionOutput(ofproto.OFPP_IN_PORT)]
    self._do_packet_out(datapath, res_pkt.data, in_port, actions)

    # wait for REPORT messages.
    hub.spawn(self._do_timeout_for_leave, timeout, datapath,
              leave.address, in_port)

```

스누핑 인스턴스에서 IGMP Report Message (Leave 응답) 수신 타임 아웃 처리

IGMP Leave Message 처리 중에서 IGMP Query Message 보낸 후 일정 시간 후에 IGMP Report Message 수신 타임 아웃 처리를 시작합니다. 멀티 캐스트 그룹 참여 호스트가 존재한다면 일반적으로 제한 시간 만료 전에 IGMP Report Message를 보내 오기 때문에 보존 영역 업데이트되지 않습니다.

일정 시간 경과 한 후에도 여전히 특정 멀티 캐스트 그룹에 대한 IGMP Report Message를 수신하지 못하면 해당 포트의 끝에는 해당 마루찌카 스트 그룹에 참여하는 호스트가 없어진 것으로 간주 멀티 캐스트 그룹 변경 이벤트의 전송 흐름 항목의 업데이트 정보 유지 영역 업데이트합니다.

포트 전송 대상에서 제외 한 결과 해당 멀티 캐스트 그룹에 참가하는 호스트가 어떤 포트 먼저 모이 않는 경우 멀티 캐스트 그룹 삭제 이벤트를 보내고, 흐름 항목의 삭제, 보존 영역의 업데이트를 수행합니다. 이 때 보유하고 있는 IGMP Leave Message가 있으면 쿼리 기에 보냅니다.

```

def _do_timeout_for_leave(self, timeout, datapath, dst, in_port):
    # ...
    hub.sleep(timeout)
    # ...

    if self._to_hosts[dpid][dst]['ports'][in_port]['out']:
        return

    del self._to_hosts[dpid][dst]['ports'][in_port]
    self._del_flow_entry(datapath, in_port, dst)

    # ...

```

```

if len(actions):
    self._send_event(
        EventMulticastGroupStateChanged(
            MG_MEMBER_CHANGED, dst, outport, ports))
    self._set_flow_entry(
        datapath, actions, outport, dst)
    self._to_hosts[dpid][dst]['leave'] = None
else:
    self._remove_multicast_group(datapath, outport, dst)
    del self._to_hosts[dpid][dst]

def _remove_multicast_group(self, datapath, outport, dst):
    # ...

    leave = self._to_hosts[dpid][dst]['leave']
    if leave:
        if ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            in_port = leave.in_port
        else:
            in_port = leave.match['in_port']
        actions = [parser.OFPActionOutput(outport)]
        self._do_packet_out(
            datapath, leave.data, in_port, actions)

```

의사 크 지역 인스턴스에서 IGMP Query Message 정기 송신 처리

의사 크 지역 인스턴스는 60 초에 한 번 IGMP Query Message 흥수합니다. 흥수 후 일정 시간이 지나면 IGMP Report Message 수신 타임 아웃 처리를 개발 시작합니다.

```

def _send_query(self):
    # ...
    timeout = 60
    # ...
    while True:
        # ...
        self._do_packet_out(
            self._datapath, res_pkt.data, send_port, flood)
        hub.sleep(igmp.QUERY_RESPONSE_INTERVAL)
        # ...

```

의사 크 지역 인스턴스에서 IGMP Report Message 처리

의사 크 지역 인스턴스는 멀티 캐스트 그룹 참여 호스트 및 스누谮 그 인스턴스에서 IGMP Report Message 를 수신했을 때 해당 멀티 캐스트 주소 스의 대상은 수신 포트가 기억되어 있지 않으면 정보를 기억하고 흐름 항목 를 등록합니다.

```

def _do_report(self, report, in_port, msg):
    # ...
    update = False
    self._mcast.setdefault(report.address, {})
    if not in_port in self._mcast[report.address]:
        update = True
    self._mcast[report.address][in_port] = True

    if update:
        actions = []
        for port in self._mcast[report.address]:
            actions.append(parser.OFPActionOutput(port))
        self._set_flow_entry(
            datapath, actions, self.server_port, report.address)

```

```
self._set_flow_entry(
    datapath,
    [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, size)],
    in_port, report.address)
```

의사 크 지역 인스턴스에서 IGMP Report Message 수신 타임 아웃 처리

IGMP Query Message 정기적 보낸 후 일정 시간 후에 IGMP Report Message 수신 시간 초과 트 프로세스를 시작합니다. IGMP Report Message가 전송되지 않은 포트에 대해서는 의사 크 지역 인스턴스는 기억 한 정보의 업데이트 및 흐름 항목 업데이트합니다. 전송 대상 포트가없는 경우, 흐름 항목 삭제합니다.

```
def _send_query(self):
    # ...
    while True:
        # ...
        hub.sleep(igmp.QUERY_RESPONSE_INTERVAL)
        # ...
        del_groups = []
        for group, status in self._mcast.items():
            del_ports = []
            actions = []
            for port in status.keys():
                if not status[port]:
                    del_ports.append(port)
                else:
                    actions.append(parser.OFPActionOutput(port))
            if len(actions) and len(del_ports):
                self._set_flow_entry(
                    self._datapath, actions, self.server_port, group)
            if not len(actions):
                self._del_flow_entry(
                    self._datapath, self.server_port, group)
            del_groups.append(group)
        if len(del_ports):
            for port in del_ports:
                self._del_flow_entry(self._datapath, port, group)
        for port in del_ports:
            del status[port]
        for group in del_groups:
            del self._mcast[group]
```

의사 크 지역 인스턴스에서 IGMP Leave Message 처리

의사 크 지역 인스턴스는 멀티 캐스트 그룹 참여 호스트에서 IGMP Leave Message를 수신했을 때, 기억 한 정보의 업데이트 및 흐름 항목 업데이트를 수행 있습니다. 전송 대상 포트가없는 경우, 흐름 항목 삭제합니다.

```
def _do_leave(self, leave, in_port, msg):
    """the process when the querier received a LEAVE message."""
    datapath = msg.datapath
    parser = datapath.ofproto_parser

    self._mcast.setdefault(leave.address, {})
    if in_port in self._mcast[leave.address]:
        self._del_flow_entry(
            datapath, in_port, leave.address)
    del self._mcast[leave.address][in_port]
    actions = []
    for port in self._mcast[leave.address]:
        actions.append(parser.OFPActionOutput(port))
```

```

if len(actions):
    self._set_flow_entry(
        datapath, actions, self.server_port, leave.address)
else:
    self._del_flow_entry(
        datapath, self.server_port, leave.address)

```

7.3.2 응용 프로그램 구현

「Ryu 응용 프로그램의 실행」에 나와있는 OpenFlow 1.3 지원 IGMP 스누핑 응용 프로그램 (simple_switch_igmp_13.py)과 「스위칭 허브」스위칭 허브의 차이를 차례로 설명합니다.

「_CONTEXTS」설정

ryu.base.app_manager.RyuApp을 계승 한 Ryu 응용 프로그램은 「_CONTEXTS」 사전에 다른 Ryu 응용 프로그램을 설정하여 다른 응용 프로그램을 별도의 스레드에서 실행시킬 수 있습니다. 여기에서는 IGMP 스누핑 라이브 리아의 IgmpLib 클래스를 「igmplib」라는 이름으로 「_CONTEXTS」로 설정합니다.

```

from ryu.lib import igmplib

# ...

class SimpleSwitchIgmp13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'igmplib': igmplib.IgmpLib}

# ...

```

「_CONTEXTS」 설정 한 응용 프로그램은 __init__() 메서드 kwargs에서 인스턴스를 얻을 수 있습니다.

```

# ...
def __init__(self, *args, **kwargs):
    super(SimpleSwitchIgmp13, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self._snoop = kwargs['igmplib']
# ...

```

라이브러리의 기본

「_CONTEXTS」로 설정 한 IGMP 스누핑 라이브러리의 초기 구성은 다음과 같습니다. 「Ryu 응용 프로그램의 실행」에서 보여준대로 쿼리 기의 동작도 가짜해야합니다. 요청이 있는 경우 IGMP 스누핑 라이브러리가 제공하는 set_querier_mode() 메서드를 실행합니다. 여기에 다음 값을 설정합니다.

매개변수	값	설명
dpid	str_to_dpid('0000000000000001')	쿼리 기 역할을 하는 데이터패스 ID
server_port	2	멀티캐스트 서버가 연결되어 있는 쿼리 기의 포트

이 설정은 데이터 경로 ID 「0000000000000001」의 OpenFlow 스위치가 쿼리 기 역할을 하며 멀티 캐스트 패킷의 원본으로 포트 2를 상정 한 후로 엔트리 등록 할 수 있습니다.

```

# ...
    self._snoop = kwargs['igmplib']
    self._snoop.set_querier_mode(
        dpid=str_to_dpid('0000000000000001'), server_port=2)
# ...

```

사용자 정의 이벤트를 수신하는 방법

「 링크 어그리게이션 」뿐만 아니라 IGMP 스누핑 라이브러리는 IGMP 파킷이 포함되지 않은 Packet-In 메시지를 EventPacketIn라는 사용자 정의 이벤트로 보냅니다. 사용자 정의 이벤트의 이벤트 처리기도 Ryu 제공하는 이벤트 처리기처럼 ryu.controller.handler.set_ev_cls 데코레이터로 장식합니다.

```
@set_ev_cls(igmplib.EventPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    # ...
```

또한 IGMP 스누핑 라이브러리는 멀티 캐스트 그룹의 추가 / 변경 / 삭제를 때 EventMulticastGroupStateChanged 이벤트를 송신하기 때문에, 이쪽 이 이벤트 처리기를 만들어 둡니다.

```
@set_ev_cls(igmplib.EventMulticastGroupStateChanged,
            MAIN_DISPATCHER)
def _status_changed(self, ev):
    msg = {
        igmplib.MG_GROUP_ADDED: 'Multicast Group Added',
        igmplib.MG_MEMBER_CHANGED: 'Multicast Group Member Changed',
        igmplib.MG_GROUP_REMOVED: 'Multicast Group Removed',
    }
    self.logger.info("%s: [%s] querier:[%s] hosts:%s",
                    msg.get(ev.reason), ev.address, ev.src,
                    ev.dsts)
```

이상과 같이, IGMP 스누핑 기능을 제공하는 라이브러리와 라이브러리를 사용하는 응용 프로그램에서 IGMP 스누핑 기능을 가진 스위칭 허브 더보기 프로그램을 실현하고 있습니다.

OpenFlow 프로토콜

이 장에서는 OpenFlow 프로토콜에 정의된 일치와 지침 및 작업에 대해 설명합니다.

8.1 매치

매치에 사용할 수 있는 조건에는 여러 가지가 있으며, OpenFlow의 버전이 올라갈 때마다 그 종류는 증가하고 있습니다. OpenFlow 1.0에서는 12 종류였지만, OpenFlow 1.3은 40 가지의 조건이 정의되어 있습니다.

개별 자세한 내용은 OpenFlow 스펙 등을 참조하시면 됩니다만, 여기에서는 OpenFlow 1.3 Match 필드를 간단하게 소개합니다.

Match 필드 이름	설명
in_port	수신 포트의 포트 번호
in_phy_port	수신 포트의 물리적 포트 번호
metadata	테이블 간에 정보를 전달하는 데 사용되는 메타 데이터
eth_dst	Ethernet 대상 MAC 주소
eth_src	Ethernet 원본 MAC 주소
eth_type	Ethernet 프레임 타입
vlan_vid	VLAN ID
vlan_pcp	VLAN PCP
ip_dscp	IP DSCP
ip_ecn	IP ECN
ip_proto	IP 프로토콜 종별
ipv4_src	IPv4의 소스 IP 주소
ipv4_dst	IPv4의 대상 IP 주소
tcp_src	TCPd 원본 포트 번호
tcp_dst	TCP 목적지 포트 번호
udp_src	UDP 소스 포트 번호
udp_dst	UDP 대상 포트 번호
sctp_src	SCTP의 원본 포트 번호
sctp_dst	SCTP 목적지 포트 번호
icmpv4_type	ICMP의 Type
icmpv4_code	ICMP의 Code
arp_op	ARP의 작동 코드
arp_spa	ARP의 소스 IP 주소
arp_tpa	ARP의 대상 IP 주소
arp_shaa	ARP의 소스 MAC 주소
arp_thaa	ARP의 대상 MAC 주소
일반 색인	

Table 8.1 – 이전 페이지에서 계속

Match 필드 이름	설명
ipv6_src	IPv6의 소스 IP 주소
ipv6_dst	IPv6의 대상 IP 주소
ipv6_flabel	IPv6의 흐름 레이블
icmpv6_type	ICMPv6의 Type
icmpv6_code	ICMPv6의 Code
ipv6_nd_target	IPv6 네이버 디스커버리의 대상 주소
ipv6_nd_sll	IPv6 네이버 디스커버리 원본 링크 계층 주소
ipv6_nd_tll	IPv6 네이버 디스커버리 타겟 링크 계층 주소
mpls_label	MPLS 레이블
mpls_tc	MPLS 트래픽 클래스 (TC)
mpls_bos	MPLS의 BoS 비트
pbb_isid	802.1ah PBB의 I-SID
tunnel_id	논리 포트에 대한 메타 데이터
ipv6_exthdr	IPv6 확장 헤더의 의사 필드

MAC 주소와 IP 주소 등의 일부 필드는 또한 마스크를 지정하는 수 있습니다.

8.2 지침

지침은 일치에 해당하는 패킷을 수신했을 때의 동작을 정의하는 것으로, 다음과 같은 유형이 규정되어 있습니다.

지침	설명
Goto Table (필수)	OpenFlow 1.1 이상에서는 여러 흐름 테이블 지원 트되고 있습니다. Goto Table 의해 일치 된 패킷의 처리를 지정된 흐름 테이블에 인수 수 있습니다. 예를 들어, “포트 1에서 받은 패킷 트에 VLAN-ID 200을 추가하여 테이블 2에 난다”라고 흐름 항목을 설정할 수 있습니다. 지정된 테이블 ID는 현재 테이블 ID보다 큰 값이 아니면 안됩니다.
Write Metadata (옵션)	이후의 테이블에서 볼 수 있는 메타 데이터를 설정합니다.
Write Actions (필수)	현재 액션 세트에 지정된 액션을 추가 가입합니다. 동일한 유형의 작업이 이미 설정된 한 경우에는 새로운 조치로 대체됩니다입니다.
Apply Actions (옵션)	액션 세트는 변경하지 않고, 지정된 액션을 즉시 적용합니다.
Clear Actions (옵션)	현재 액션 세트의 모든 액션을 제거 합니다.
Meter (옵션)	지정된 미터에 패킷을 적용합니다.

Ryu는 각 명령어에 해당하는 다음의 클래스가 구현되어 있습니다.

- OFPInstructionGotoTable
- OFPInstructionWriteMetadata
- OFPInstructionActions
- OFPInstructionMeter

Write/Apply/Clear Actions는 OFPInstructionActions에 정리하고 있고, 인스턴스 생성 시에 선택합니다.

주의: Write Actions의 지원은 필수로 되어 있습니다만, 현재의 Open vSwitch에서 지원되지 않습니다. Apply Actions를 지원하고 있으므로, 대신이 옵션을 사용할 필요가 있습니다. Write Actions는 Open vSwitch 2.1.0에서 지원 될 예정입니다.

8.3 액션

OFPActionOutput 클래스는 Packet-Out 메시지와 Flow Mod 메시지 사용 패킷 전송을 지정하는 것입니다. 생성자의 인수 대상과 컨트롤러에 보내려면 최대 데이터 크기 (max_len)을 지정합니다. 대상에는 스위치

의 물리적 포트 번호 외에 몇 가지 정의 된 값이 지정할 수 있습니다.

값	설명
OFPP_IN_PORT	수신 포트로 전송됩니다
OFPP_TABLE	위로의 흐름 테이블에 적용됩니다
OFPP_NORMAL	스위치의 L2/L3 기능으로 전송됩니다
OFPP_FLOOD	수신 포트 또는 블록 된 포트를 제외한 해당 VLAN의 모든 물리적 포트에 Flooding
OFPP_ALL	수신 포트를 제외한 모든 물리적 포트에 전송됩니다
OFPP_CONTROLLER	컨트롤러에 Packet-In 메시지로 보내집니다
OFPP_LOCAL	스위치의 로컬 포트를 지정합니다
OFPP_ANY	Flow Mod (delete) 메시지 및 Flow Stats Requests 메시지에서 포트를 선택할 때 와일드 카드로 사용하는 것으로, 패킷 전송에서 사용되지 않습니다

max_len 0을 지정하면 Packet-In 메시지 패킷의 이진 데이터 첨부 된 없습니다. OFPCML_NO_BUFFER 을 지정하면 OpenFlow 스위치에서 패킷을 버퍼없이 Packet-In 메시지 패킷 전체가 첨부됩니다.

ofproto 라이브러리

이 장에서는 Ryu의 ofproto 라이브러리에 대해 소개합니다.

9.1 개요

ofproto 라이브러리는 OpenFlow 프로토콜 메시지의 작성 · 분석을 행하기 위한 라이브러리입니다.

9.2 모듈 구성

각 OpenFlow 버전 (버전 XY)에 대해 상수 모듈 (ofproto_vX_Y)과 파서 모듈 (ofproto_vX_Y_parser)가 포함되어 있습니다. 각 OpenFlow 버전의 구현은 기본적으로 독립되어 있습니다.

OpenFlow 버전	상수 모듈	파서 모듈
1.0.x	ryu.ofproto.ofproto_v1_0	ryu.ofproto.ofproto_v1_0_parser
1.2.x	ryu.ofproto.ofproto_v1_2	ryu.ofproto.ofproto_v1_2_parser
1.3.x	ryu.ofproto.ofproto_v1_3	ryu.ofproto.ofproto_v1_3_parser
1.4.x	ryu.ofproto.ofproto_v1_4	ryu.ofproto.ofproto_v1_4_parser

9.2.1 상수 모듈

상수 모듈은 프로토콜 상수 정의합니다. 예를 들면 다음과 같다.

상수	설명
OFP_VERSION	프로토콜 버전 번호
OFPP_xxxx	포트 번호
OFPCML_NO_BUFFER	버퍼 없이 전체 패킷을 전송
OFP_NO_BUFFER	잘못된 버퍼 번호

9.2.2 파서 모듈

파서 모듈은 각 OpenFlow 메시지에 대응 한 클래스가 정의되어 있습니다. 예를 들면 다음과 같다. 이 클래스와 그 인스턴스를 앞으로 메시지 클래스 메시지 개체라고 합니다.

클래스	설명
OFPHello	OFPT_HELLO 메시지
OFPPacketOut	OFPT_PACKET_OUT 메시지
OFPFlowMod	OFPT_FLOW_MOD 메시지

또한 파서 모듈은 OpenFlow 메시지의 페이로드 중에 사용되는 구조에 대응하는 클래스도 정의되어 있습니다. 예를 들면 다음과 같다. 이 클래스와 그 인스턴스를 향후 구조 클래스 자체라고 합니다.

클래스	구조체
OFPMatch	ofp_match
OFPInstructionGotoTable	ofp_instruction_goto_table
OFPActionOutput	ofp_action_output

9.3 기본적인 사용법

9.3.1 ProtocolDesc 클래스

사용하는 OpenFlow 프로토콜을 지정하기 위한 클래스입니다. 메시지 클래스의 `__init__`의 `datapath` 인수는 이 클래스(또는 파생 클래스인 Datapath 클래스)의 객체를 지정합니다.

```
from ryu.ofproto import ofproto_protocol
from ryu.ofproto import ofproto_v1_3

dp = ofproto_protocol.ProtocolDesc(version=ofproto_v1_3.OFP_VERSION)
```

9.3.2 네트워크 주소

Ryu ofproto 라이브러리의 API는 기본적으로 문자열 표현의 네트워크 주소가 사용됩니다. 예를 들면 다음과 같다.

주석: 그러나 OpenFlow 1.0에 관해서는 다른 표현이 사용되고 있습니다. (2014년 2월 현재)

주소 종류	python 문자열 예제
MAC 주소	'00:03:47:8c:a1:b3'
IPv4 주소	'192.0.2.1'
IPv6 주소	'2001:db8::2'

9.3.3 메시지 개체의 생성

각 메시지 클래스, 구조체 클래스의 인스턴스를 적절한 인수로 생성합니다.

인수의 이름은 기본적으로 OpenFlow 프로토콜에서 정해진 필드 이름 동일합니다. 그러나 python의 예약어와 충돌하는 경우 마지막에 `__`를 넣습니다. 다음 예제에서는 `type_`이 이에 해당됩니다.

```
from ryu.ofproto import ofproto_protocol
from ryu.ofproto import ofproto_v1_3

dp = ofproto_protocol.ProtocolDesc(version=ofproto_v1_3.OFP_VERSION)
ofp = dp.ofproto
ofpp = dp.ofproto_parser
actions = [parser.OFPActionOutput(port=ofp.OFPP_CONTROLLER,
                                    max_len=ofp.OFPCML_NO_BUFFER)]
inst = [parser.OFPInstructionActions(type_=ofp.OFPIT_APPLY_ACTIONS,
                                       actions=actions)]
fm = ofpp.OFPPFlowMod(datapath=dp,
                       priority=0,
                       match=ofpp.OFPMatch(in_port=1,
                                           eth_src='00:50:56:c0:00:08'),
                       instructions=inst)
```

주석: 상수 모듈 파서 모듈은 직접 import하여 사용해도 좋지만, 사용하는 OpenFlow 버전을 변경할 때 최소한의 수정으로 끝나도록, 가능한 ProtocolDesc 개체의 ofproto, ofproto_parser 특성을 사용하는 것을 권장합니다.

9.3.4 메시지 개체의 분석

메시지 개체의 내용을 확인할 수 있습니다.

예를 들어 OFPPacketIn 개체 pid의 match 필드가 pin.match로 액세스 할 수 있습니다.

OFPMatch 개체의 각 TLV 다음과 같이 이름으로 액세스 할 수 있습니다.

```
print pin.match['in_port']
```

9.3.5 JSON

메시지 개체를 json.dumps 호환 사전으로 변환하는 기능과 json.loads 호환 사전에서 메시지 개체를 복원하는 기능이 있습니다.

주석: 그러나 OpenFlow 1.0 관해서는 구현이 불완전합니다. (2014년 2월 현재)

```
import json

print json.dumps(msg.to_jsondict())
```

9.3.6 메시지의 해석 (Parse)

메시지의 바이트 열에서 해당 메시지 객체를 생성합니다. 스위치에서 받은 메시지 내용은 프레임 워크가 자동으로 이 처리를 행하기 위해, Ryu 응용 프로그램이 의식 할 필요는 없습니다.

구체적으로는 다음과 같습니다.

1. ryu.ofproto.ofproto_parser.header 함수를 사용하여 버전 독립적 부분을 분석
2. ryu.ofproto.ofproto_parser.msg 함수에 전달하여 나머지 부분을 분석

9.3.7 메시지의 생성 (연재)

메시지 개체에서 해당 메시지의 바이트를 생성합니다. 스위치에 보내는 메시지 내용은 프레임 워크가 자동으로 이 처리를 행하기 위해, Ryu 응용 프로그램이 의식 할 필요는 없습니다.

구체적으로는 다음과 같습니다.

1. 메시지 개체의 serialize 메소드를 호출
2. 메시지 개체의 buf 특성을 읽을

'len' 같은 일부 필드는 명시 적으로 값을 지정하지 않아도 serialize시 자동으로 계산됩니다.

패킷 라이브러리

OpenFlow의 Packet-In과 Packet-Out 메시지는 원시 패킷 내용을 나타내는 바이트가 들어가는 필드가 있습니다. Ryu에는 이러한 원시 패킷을 응용 프로그램에서 다루기 쉽고 하는 라이브러리가 포함되어 있습니다. 이 장에서는 이 라이브러리를 소개합니다.

10.1 기본적인 사용법

10.1.1 프로토콜 헤더 클래스

Ryu 패킷 라이브러리는 다양한 프로토콜 헤더에 대응하는 클래스가 포함되어 있습니다.

다음을 포함 프로토콜을 지원합니다. 각 프로토콜에 대응하는 클래스 등의 자세한 것은 [API 레퍼런스](#)를 참조하십시오.

- arp
- bgp
- bpdu
- dhcp
- ethernet
- icmp
- icmpv6
- igmp
- ipv4
- ipv6
- llc
- lldp
- mpls
- ospf
- pbb
- sctp
- slow
- tcp

- udp
- vlan
- vrrp

각 프로토콜 헤더 클래스의 `__init__` 인수 이름은 기본적으로 RFC 등 사용 된 이름과 동일하게되어 있습니다. 프로토콜 헤더 클래스의 인스턴스 속성의 명명 규칙도 마찬가지입니다. 그러나 `type` 등 Python built-in 과 충돌하는 이름의 필드에 해당하는 `__init__` 인수 이름은 `type_`처럼 마지막에 `_`가 붙습니다.

일부 `__init__` 인수는 기본값이 설정되어 생략 할 수 있습니다. 다음 예제에서는 `version=4` 등이 생략되어 있습니다.

```
from ryu.lib.ofproto import inet
from ryu.lib.packet import ipv4

pkt_ipv4 = ipv4.ipv4(dst='192.0.2.1',
                     src='192.0.2.2',
                     proto=inet.IPPROTO_UDP)

print pkt_ipv4.dst
print pkt_ipv4.src
print pkt_ipv4.proto
```

10.1.2 네트워크 주소

Ryu 패킷 라이브러리의 API는 기본적으로 문자열 표현의 네트워크 주소가 사용됩니다. 예를 들면 다음과 같습니다.

주소 종류	python 문자열 예제
MAC 주소	'00:03:47:8c:a1:b3'
IPv4 주소	'192.0.2.1'
IPv6 주소	'2001:db8::2'

10.1.3 패킷 분석 (Parse)

패킷의 바이트 열에서 해당 python 객체를 생성합니다.

구체적으로는 다음과 같습니다.

1. `ryu.lib.packet.packet.Packet` 클래스의 객체를 생성 (data 인수 분석하는 바이트를 지정)
2. 1. 개체의 `get_protocol` 메서드 등을 사용하여 각 프로토콜 헤더에 해당하는 개체를 가져

```
pkt = packet.Packet(data=bin_packet)
pkt_ether = pkt.get_protocol(ethernet.ethernet)
if not pkt_ether:
    # non ethernet
    return
print pkt_ether.dst
print pkt_ether.src
print pkt_ether.ethertype
```

10.1.4 패킷의 생성 (연재)

python 객체에서 해당 패킷의 바이트를 생성합니다.

구체적으로는 다음과 같습니다.

1. `ryu.lib.packet.packet.Packet` 클래스의 객체를 생성
2. 각 프로토콜 헤더에 해당하는 객체를 생성 (ethernet, ipv4, ...)

3. (a) 개체의 add_protocol 메서드를 사용하여 2. 헤더를 차례로 추가

4. (a) 개체 serialize 메서드를 호출하여 결과 바이트를

체크섬과 페이로드 길이 등의 일부 필드는 명시 적으로 값을 지정하지 않아도 serialize시 자동으로 계산됩니다. 자세한 내용은 각 클래스 참조를 참조하십시오.

```
pkt = packet.Packet()
pkt.add_protocol(ethernet.ethernet(ethertype=...,
                                    dst=...,
                                    src=...))
pkt.add_protocol(ipv4.ipv4(dst=...,
                           src=...,
                           proto=...))
pkt.add_protocol(icmp.icmp(type_=...,
                           code=...,
                           csum=...,
                           data=...))

pkt.serialize()
bin_packet = pkt.data
```

Scapy 좋아하는 대체 API도 포함되어 있기 때문에 취향에 따라 사용해주십시오.

```
e = ethernet.ethernet(...)
i = ipv4.ipv4(...)
u = udp.udp(...)
pkt = e/i/u
```

10.2 어플리케이션 예

위의 예제를 사용하여 만든 ping에 응답하는 응용 프로그램을 보여줍니다.

ARP REQUEST와 ICMP ECHO REQUEST를 Packet-In에서 받아 답장을 Packet-Out으로 보냅니다. IP 주소 등은 __init__ 메서드에 하드 코드되어 있습니다.

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
# Copyright (C) 2013 YAMAMOTO Takashi <yamamoto at valinux co jp>
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# a simple ICMP Echo Responder

from ryu.base import app_manager

from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls

from ryu.ofproto import ofproto_v1_3

from ryu.lib.packet import packet
```

```

from ryu.lib.packet import ethernet
from ryu.lib.packet import arp
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp

class IcmpResponder(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(IcmpResponder, self).__init__(*args, **kwargs)
        self.hw_addr = '0a:e4:1c:d1:3e:44'
        self.ip_addr = '192.0.2.9'

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def _switch_features_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        actions = [parser.OFPActionOutput(port=ofproto.OFPP_CONTROLLER,
                                           max_len=ofproto.OFPCML_NO_BUFFER)]
        inst = [parser.OFFPInstructionActions(type_=ofproto.OFPIT_APPLY_ACTIONS,
                                              actions=actions)]
        mod = parser.OFPFlowMod(datapath=datapath,
                               priority=0,
                               match=parser.OFPMatch(),
                               instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        port = msg.match['in_port']
        pkt = packet.Packet(data=msg.data)
        self.logger.info("packet-in %s" % (pkt,))
        pkt_ether = pkt.get_protocol(ether.ethernet)
        if not pkt_ether:
            return
        pkt_arp = pkt.get_protocol(arp.arp)
        if pkt_arp:
            self._handle_arp(datapath, port, pkt_ether, pkt_arp)
            return
        pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
        pkt_icmp = pkt.get_protocol(icmp.icmp)
        if pkt_icmp:
            self._handle_icmp(datapath, port, pkt_ether, pkt_ipv4, pkt_icmp)
            return

    def _handle_arp(self, datapath, port, pkt_ether, pkt_arp):
        if pkt_arp.opcode != arp.ARP_REQUEST:
            return
        pkt = packet.Packet()
        pkt.add_protocol(ether.ethernet(ethertype=pkt_ether.ethertype,
                                       dst=pkt_ether.src,
                                       src=self.hw_addr))
        pkt.add_protocol(arp.arp(opcode=arp.ARP_REPLY,
                               src_mac=self.hw_addr,
                               src_ip=self.ip_addr,
                               dst_mac=pkt_arp.src_mac,
                               dst_ip=pkt_arp.src_ip))
        self._send_packet(datapath, port, pkt)

```

```

def _handle_icmp(self, datapath, port, pkt_ether, pkt_ip, pkt_icmp):
    if pkt_icmp.type != icmp.ICMP_ECHO_REQUEST:
        return
    pkt = packet.Packet()
    pkt.add_protocol(ether.ethernet(ethertype=pkt_ether.ethertype,
                                    dst=pkt_ether.src,
                                    src=self.hw_addr))
    pkt.add_protocol(ipv4.ipv4(dst=pkt_ip.src,
                               src=self.ip_addr,
                               proto=pkt_ip.proto))
    pkt.add_protocol(icmp.icmp(type_=icmp.ICMP_ECHO_REPLY,
                               code=icmp.ICMP_ECHO_REPLY_CODE,
                               csum=0,
                               data=pkt_icmp.data))
    self._send_packet(datapath, port, pkt)

def _send_packet(self, datapath, port, pkt):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    pkt.serialize()
    self.logger.info("packet-out %s" % (pkt,))
    data = pkt.data
    actions = [parser.OFPActionOutput(port=port)]
    out = parser.OFPPacketOut(datapath=datapath,
                              buffer_id=ofproto.OFP_NO_BUFFER,
                              in_port=ofproto.OFPP_CONTROLLER,
                              actions=actions,
                              data=data)
    datapath.send_msg(out)

```

주석: OpenFlow 1.2 이상에서는 Packet-In 메시지 match 필드에서 퍼스 된 패킷 헤더의 내용을 검색 할 수 있습니다. 그러나 이 필드에 얼마나 많은 정보를 넣어 줄까 스위치의 구현에 따라 다릅니다. 예를 들어 Open vSwitch는 최소한의 정보만 넣어주지 않으므로 많은 경우 컨트롤러 측에서 패킷 내용을 분석해야합니다. 한편 LINC는 가능한 한 많은 정보를 넣어줍니다.

다음은 ping -c 3를 실행 한 경우 로그의 예입니다

```

EVENT ofp_event->IcmpResponder EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xb63c802c OFPSwitchFeatures(auxiliary_id=0, capabili
move onto main mode
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='ff:ff:ff:ff:ff:ff', ethertype=2054, src='0a:e4:1c:d1:3e:43'), arp(dst_ip='192.168.1.100', src_ip='192.168.1.101')
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2054, src='0a:e4:1c:d1:3e:44'), arp(dst_ip='192.168.1.101', src_ip='192.168.1.100')
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44', ethertype=2048, src='0a:e4:1c:d1:3e:43'), ipv4(csum=47312, dst_ip='192.168.1.100', src_ip='192.168.1.101')
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2048, src='0a:e4:1c:d1:3e:44'), ipv4(csum=14160, dst_ip='192.168.1.101', src_ip='192.168.1.100')
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44', ethertype=2048, src='0a:e4:1c:d1:3e:43'), ipv4(csum=47312, dst_ip='192.168.1.100', src_ip='192.168.1.101')
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2048, src='0a:e4:1c:d1:3e:44'), ipv4(csum=14160, dst_ip='192.168.1.101', src_ip='192.168.1.100')
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44', ethertype=2048, src='0a:e4:1c:d1:3e:43'), ipv4(csum=47312, dst_ip='192.168.1.100', src_ip='192.168.1.101')
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2048, src='0a:e4:1c:d1:3e:44'), ipv4(csum=14160, dst_ip='192.168.1.101', src_ip='192.168.1.100')

```

IP 조각 대응은 독자에게 숙제로 합니다. OpenFlow 프로토콜 자체에는 MTU를 검색하는 방법이 없기 때문에, 하드 코딩하거나 어떤 궁리가 필요합니다. 또한 Ryu 패킷 라이브러리는 항상 패킷 전체 퍼스 / 연재 때문에 단편화 된 패킷을 처리하기 위한 API 변경이 필요합니다.

OF-Config 라이브러리

이 장에서는 Ryu에 포함 된 OF-Config 클라이언트 라이브러리에 대해 소개합니다.

11.1 OF-Config 프로토콜

OF-Config는 OpenFlow 스위치의 관리를 위한 프로토콜입니다. NETCONF (RFC 6241) 스키마로 정의되어며, 논리 스위치, 포트, 큐 등의 상태 취득이나 설정을 할 수 있습니다.

OpenFlow와 같은 ONF가 개발 한 것으로, 다음 사이트에서 사양이 사용할 수 있습니다.

<https://www.opennetworking.org/sdn-resources/onf-specifications/openflow-config>

이 라이브러리는 OF-Config 1.1.1을 준수하고 있습니다.

주석: 현재 Open vSwitch는 OF-Config를 지원하지 않지만 같은 목적을 위해 OVSDB하는 서비스를 제공하고 있습니다. OF-Config 비교적 새로운 표준으로 Open vSwitch가 OVSDB를 구현했을 때 아직 존재하지 않았습니다.

OVSDB 프로토콜은 RFC 7047으로 사양이 공개되어 있습니다만, 사실상 Open vSwitch 전용 프로토콜이되고 있습니다. OF-Config는 아직 등장에서 일천하지만 미래에 많은 OpenFlow 스위치에서 구현 될 것으로 예상됩니다.

11.2 라이브러리 구성

11.2.1 ryu.lib.of_config.capable_switch.OFCapableSwitch 클래스

NETCONF 세션을 처리하기 위한 클래스입니다.

```
from ryu.lib.of_config.capable_switch import OFCapableSwitch
```

11.2.2 ryu.lib.of_config.classes 모듈

설정 내용을 python 객체로 취급하기 위한 클래스 군을 제공하는 모듈입니다.

주석: 클래스 이름은 기본적으로 OF-Config 1.1.1 yang specification에 grouping 키워드로 사용되는 이름과 동일합니다.
예. OFPortType

```
import ryu.lib.of_config.classes as ofc
```

11.3 예제

11.3.1 스위치에 연결

SSH 트랜스 포트를 사용하여 스위치에 연결합니다. `unknown_host_cb`에는 알 수 없는 SSH 호스트 키를 처리하는 콜백 함수를 입니다만, 여기에서는 무조건 연결을 계속하도록하고 있습니다.

```
sess = OFCapableSwitch(
    host='localhost',
    port=1830,
    username='linc',
    password='linc',
    unknown_host_cb=lambda host, fingerprint: True)
```

11.3.2 GET

NETCONF GET을 사용하여 상태를 가져 오는 방법입니다. 모든 포트 `/resources/port/resource-id`와 `/resources/port/current-rate`를 표시합니다.

```
csw = sess.get()
for p in csw.resources.port:
    print p.resource_id, p.current_rate
```

11.3.3 GET-CONFIG

NETCONF GET-CONFIG를 사용하여 설정을 검색하는 예입니다.

주석: `running`하는 것은 NETCONF의 데이터 저장소에서 현재 운영하고있는 설정입니다. 구현에 따라, 그 밖에도 `startup` (장치를 시작할 때로드되는 기타 설정) 나 `candidate` (후보 설정) 등의 데이터 스토어를 이용할 수 있습니다.

모든 포트 `/resources/port/resource-id`와 `/resources/port/configuration/admin-state`를 표시합니다.

```
csw = sess.get_config('running')
for p in csw.resources.port:
    print p.resource_id, p.configuration.admin_state
```

11.3.4 EDIT-CONFIG

NETCONF EDIT-CONFIG를 사용하여 설정을 변경하는 예입니다. 기본적으로 GET-CONFIG에서 얻은 설정을 편집하여 EDIT-CONFIG에서 반송라는 단계입니다.

주석: 프로토콜상은 EDIT-CONFIG 설정 부분적인 편집을 할 수도 수 있지만 이러한 방법이 무난합니다.

모든 포트 `/resources/port/configuration/admin-state`를 `down`으로 설정합니다.

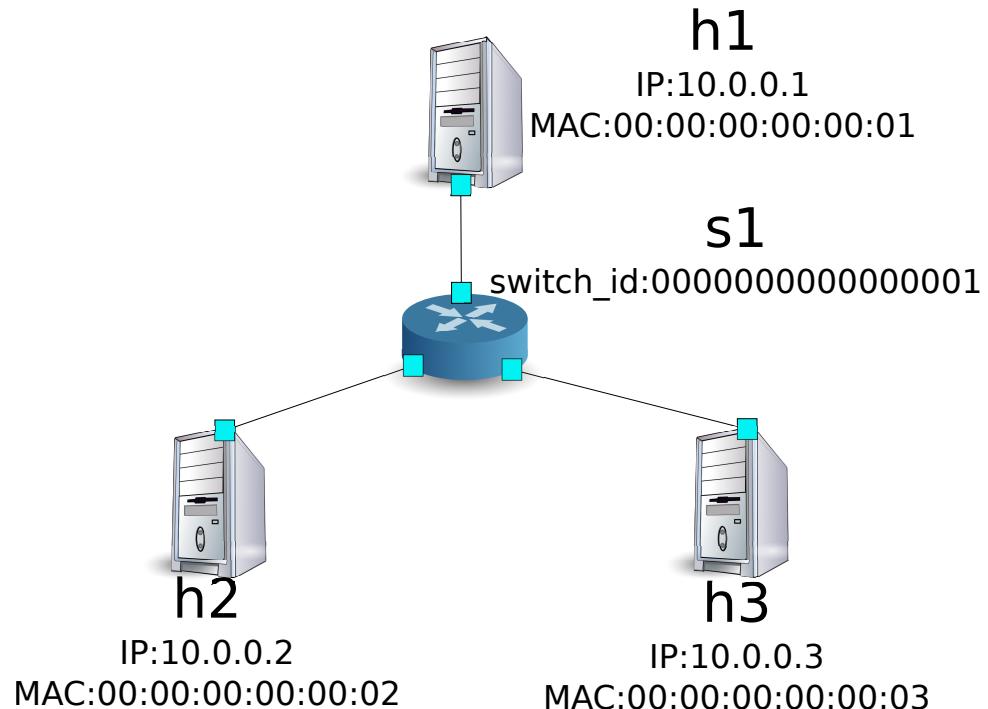
```
csw = sess.get_config('running')
for p in csw.resources.port:
    p.configuration.admin_state = 'down'
sess.edit_config('running', csw)
```

방화벽

이 장에서는 REST에서 설정을 할 수 방화벽을 사용하는 방법에 대해 설명 합니다.

12.1 단일 거주자의 동작 예

다음과 같은 토폴로지를 만들고 해당 스위치 s1에 룰 르의 추가 . 삭제하는 예를 소개합니다.



12.1.1 환경 구축

우선 Mininet에 환경을 구축합니다. 입력 명령 「스위칭 허브」이라고 같습니다.

```

ryu@ryu-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
  
```

```
*** Adding switches:  
s1  
*** Adding links:  
(h1, s1) (h2, s1) (h3, s1)  
*** Configuring hosts  
h1 h2 h3  
*** Running terms on localhost:10.0  
*** Starting controller  
*** Starting 1 switches  
s1  
  
*** Starting CLI:  
mininet>
```

또한 컨트롤러의 xterm을 또 시작해야합니다.

```
mininet> xterm c0  
mininet>
```

이어 사용하는 OpenFlow 버전을 1.3으로 설정합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

마지막으로, 컨트롤러 xterm에서 rest_firewall을 시작합니다.

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_firewall  
loading app ryu.app.rest_firewall  
loading app ryu.controller.ofp_handler  
instantiating app None of DPSet  
creating context dpset  
creating context wsgi  
instantiating app ryu.app.rest_firewall of RestFirewallAPI  
instantiating app ryu.controller.ofp_handler of OFPHandler  
(2210) wsgi starting up on http://0.0.0.0:8080/
```

Ryu와 스위치 간의 연결에 성공하면 다음 메시지가 표시됩니다.

controller: c0 (root):

```
[FW] [INFO] switch_id=0000000000000001: Join as firewall
```

12.1.2 초기 상태의 변경

firewall 시작 직후에는 모든 통신을 차단하도록 비활성화 상태로되어 있습니다. 다음 명령으로 활성화(enable)합니다.

주석: 이후의 설명에서 사용하는 REST API의 자세한 내용은 장 뒷부분의 「REST API 목록」을 참조 하십시오.

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/module/enable/0000000000000001  
[  
  {  
    "switch_id": "0000000000000001",  
    "command_result": {  
      "result": "success",  
      "details": "firewall running."  
    }  
  }  
]
```

```
root@ryu-vm:~# curl http://localhost:8080/firewall/module/status
[
  {
    "status": "enable",
    "switch_id": "00000000000000000001"
  }
]
```

주석: REST 명령의 실행 결과는 보기 쉽도록 형성하고 있습니다.

h1에서 h2에 ping 소통을 확인하여보십시오. 그러나 권한 규칙을 설정하지 않기 때문에 차단되어 버립니다.

host: h1:

```
root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
20 packets transmitted, 0 received, 100% packet loss, time 19003ms
```

차단 된 패킷 로그에 기록됩니다.

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:02', ethertype=2048)
```

12.1.3 규칙 추가

h1과 h2 사이에서 ping을 허용하는 규칙을 추가합니다. 양방향 규칙을 추가해야합니다.

다음 규칙을 추가하여 봅시다. 규칙 ID는 자동 번호 지정됩니다.

원본	대상	프로토콜	여부	(규칙ID)
10.0.0.1/32	10.0.0.2/32	ICMP	허용	1
10.0.0.2/32	10.0.0.1/32	ICMP	허용	2

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.1/32", "nw_dst": "10.0.0.2/32", "nw_proto": "ICMP", "nw_src_port": 0, "nw_dst_port": 0, "action": "allow", "switch_id": "0000000000000001", "rule_id": 1}'
{
  "command_result": [
    {
      "result": "success",
      "details": "Rule added. : rule_id=1"
    }
  ]
}

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.1/32", "nw_proto": "ICMP", "nw_src_port": 0, "nw_dst_port": 0, "action": "allow", "switch_id": "0000000000000001", "rule_id": 2}'
{
  "command_result": [
    {
      "result": "success",
      "details": "Rule added. : rule_id=2"
    }
  ]
}
```

```

    }
]
```

추가 규칙이 흐름 항목으로 스위치에 등록됩니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=823.705s, table=0, n_packets=10, n_bytes=420, priority=65534,arp actions=NONE
cookie=0x0, duration=542.472s, table=0, n_packets=20, n_bytes=1960, priority=0 actions=CONTROLLER,OUTPUT:1
cookie=0x1, duration=145.05s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src=10.0.0.1,nw_dst=10.0.0.2
cookie=0x2, duration=118.265s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src=10.0.0.2,nw_dst=10.0.0.1
```

또한 h2와 h3 사이에서 ping을 포함한 모든 IPv4 패킷을 허용하도록 규칙을 추가합니다.

원본	대상	프로토콜	여부	(규칙ID)
10.0.0.2/32	10.0.0.3/32	any	허용	3
10.0.0.3/32	10.0.0.2/32	any	허용	4

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32"}' http://localhost:8080/ofctl/flows
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Rule added. : rule_id=3"
      }
    ]
  }
]

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32"}' http://localhost:8080/ofctl/flows
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Rule added. : rule_id=4"
      }
    ]
  }
]
```

추가 규칙이 흐름 항목으로 스위치에 등록됩니다.

switch: s1 (root):

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x3, duration=12.724s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_src=10.0.0.2,nw_dst=10.0.0.3
cookie=0x4, duration=3.668s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_src=10.0.0.3,nw_dst=10.0.0.2
cookie=0x0, duration=1040.802s, table=0, n_packets=10, n_bytes=420, priority=65534,arp actions=NONE
cookie=0x0, duration=759.569s, table=0, n_packets=20, n_bytes=1960, priority=0 actions=CONTROLLER,OUTPUT:1
cookie=0x1, duration=362.147s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src=10.0.0.1,nw_dst=10.0.0.2
cookie=0x2, duration=335.362s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src=10.0.0.2,nw_dst=10.0.0.1
```

규칙에 우선 순위를 설정할 수 있습니다.

h2와 h3 사이에서 ping (ICMP)을 차단하는 규칙을 추가 봅시다. 우선 순위로 디폴트 값 1보다 큰 값을 설정합니다.

(우선순위)	원본	대상	프로토콜	여부	(규칙ID)
10	10.0.0.2/32	10.0.0.3/32	ICMP	차단	5
10	10.0.0.3/32	10.0.0.2/32	ICMP	차단	6

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32", "nw_proto": "icmp", "priority": 10, "action": "deny"}'
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=5"}]}]

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32", "nw_proto": "icmp", "priority": 10, "action": "deny"}'
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=6"}]}]
```

추가 규칙이 흐름 항목으로 스위치에 등록됩니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPTST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x3, duration=242.155s, table=0, n_packets=0, n_bytes=0, priority=1, ip, nw_src=10.0.0.2, nw_dst=10.0.0.3, nw_proto=icmp
cookie=0x4, duration=233.099s, table=0, n_packets=0, n_bytes=0, priority=1, ip, nw_src=10.0.0.3, nw_dst=10.0.0.2, nw_proto=icmp
cookie=0x0, duration=1270.233s, table=0, n_packets=10, n_bytes=420, priority=65534, arp actions=NO_ACTION
cookie=0x0, duration=989s, table=0, n_packets=20, n_bytes=1960, priority=0 actions=CONTROLLER:127
cookie=0x5, duration=26.984s, table=0, n_packets=0, n_bytes=0, priority=10, icmp, nw_src=10.0.0.2, nw_dst=10.0.0.3, nw_proto=icmp
cookie=0x1, duration=591.578s, table=0, n_packets=0, n_bytes=0, priority=1, icmp, nw_src=10.0.0.1, nw_dst=10.0.0.2, nw_proto=icmp
cookie=0x6, duration=14.523s, table=0, n_packets=0, n_bytes=0, priority=10, icmp, nw_src=10.0.0.3, nw_dst=10.0.0.2, nw_proto=icmp
cookie=0x2, duration=564.793s, table=0, n_packets=0, n_bytes=0, priority=1, icmp, nw_src=10.0.0.2, nw_dst=10.0.0.3, nw_proto=icmp
```

12.1.4 규칙 확인

설정된 규칙을 확인합니다.

Node: c0 (root):

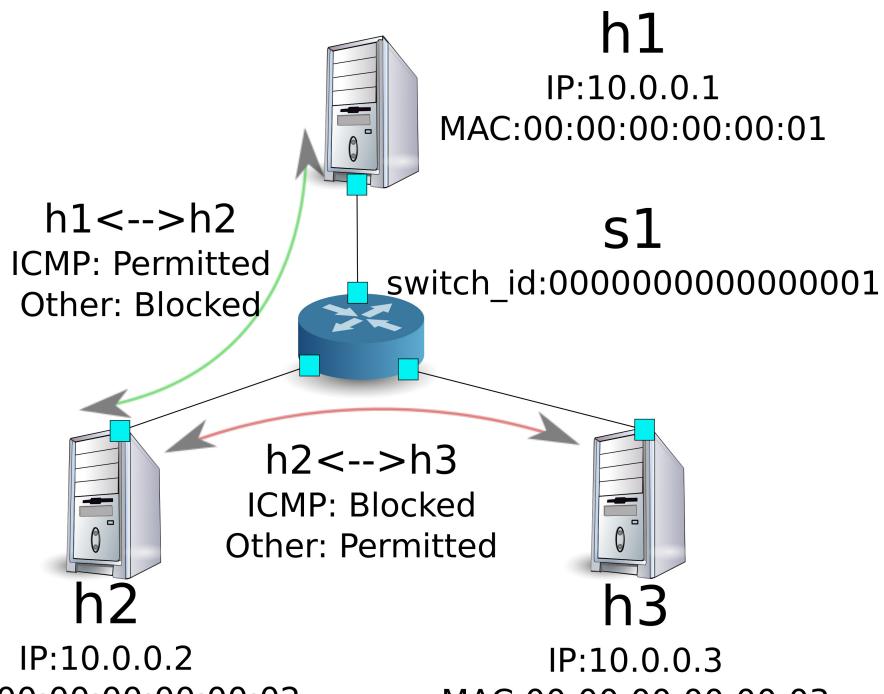
```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/0000000000000000
[{"access_control_list": [{"rules": [{"priority": 1, "dl_type": "IPv4", "nw_dst": "10.0.0.3", "action": "deny"}]}]}
```

```

        "nw_src": "10.0.0.2",
        "rule_id": 3,
        "actions": "ALLOW"
    },
    {
        "priority": 1,
        "dl_type": "IPv4",
        "nw_dst": "10.0.0.2",
        "nw_src": "10.0.0.3",
        "rule_id": 4,
        "actions": "ALLOW"
    },
    {
        "priority": 10,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "nw_dst": "10.0.0.3",
        "nw_src": "10.0.0.2",
        "rule_id": 5,
        "actions": "DENY"
    },
    {
        "priority": 1,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "nw_dst": "10.0.0.2",
        "nw_src": "10.0.0.1",
        "rule_id": 1,
        "actions": "ALLOW"
    },
    {
        "priority": 10,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "nw_dst": "10.0.0.2",
        "nw_src": "10.0.0.3",
        "rule_id": 6,
        "actions": "DENY"
    },
    {
        "priority": 1,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "nw_dst": "10.0.0.1",
        "nw_src": "10.0.0.2",
        "rule_id": 2,
        "actions": "ALLOW"
    }
]
}
],
"switch_id": "0000000000000001"
}
]

```

설정 한 규칙을 그림으로 표시하면 다음과 같습니다.



h1에서 h2로 ping을 실행 해 봅니다. 허용하는 규칙이 설정되어 있기 때문에 ping이 소통 합니다.

host: h1:

```
root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.419 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.060 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.033 ms
...
```

h1에서 h2에 ping 아닌 패킷은 firewall에 의해 차단됩니다. 예를 들어 h1에서 h2에 wget을 실행하면 패킷이 차단되었다는 로그가 출력됩니다.

host: h1:

```
root@ryu-vm:~# wget http://10.0.0.2
--2013-12-16 15:00:38--  http://10.0.0.2/
Connecting to 10.0.0.2:80... ^C
```

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet (dst='00:00:00:00:00:02', ethertype=2048)
```

h2와 h3 동안 ping 아닌 패킷의 소통이 가능 해지고 있습니다. 예를 들어 h2에서 h3에 ssh를 실행하면 패킷이 차단되었다는 로그는 출력되지 않습니다 (h3에서 sshd가 작동하지 않기 때문에 ssh에서 연결에 실패합니다).

host: h2:

```
root@ryu-vm:~# ssh 10.0.0.3
ssh: connect to host 10.0.0.3 port 22: Connection refused
```

h2에서 h3를 ping하면 패킷이 firewall에 의해 차단되었다는 로그가 출력됩니다.

host: h2:

```
root@ryu-vm:~# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
```

```

^C
--- 10.0.0.3 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7055ms

controller: c0 (root):
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:03', ethertype=2048)
...

```

12.1.5 규칙 삭제

“rule_id:5” 및 “rule_id:6” 규칙을 삭제합니다.

Node: c0 (root):

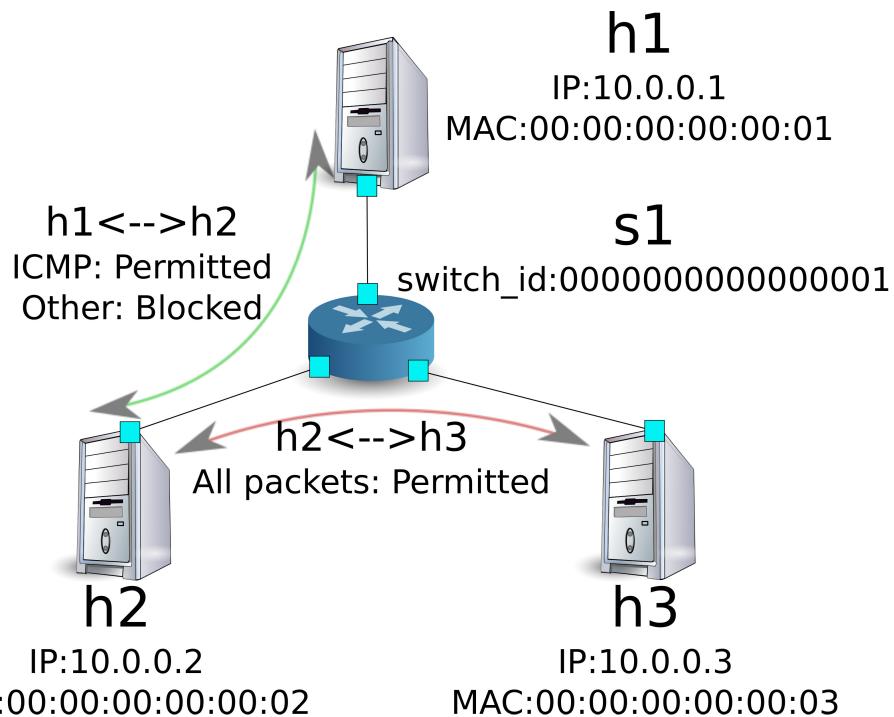
```

root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "5"}' http://localhost:8080/firewall/rules/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Rule deleted. : ruleID=5"
      }
    ]
  }
]

root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "6"}' http://localhost:8080/firewall/rules/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Rule deleted. : ruleID=6"
      }
    ]
  }
]

```

현재 규칙을 도식화하면 다음과 같습니다.



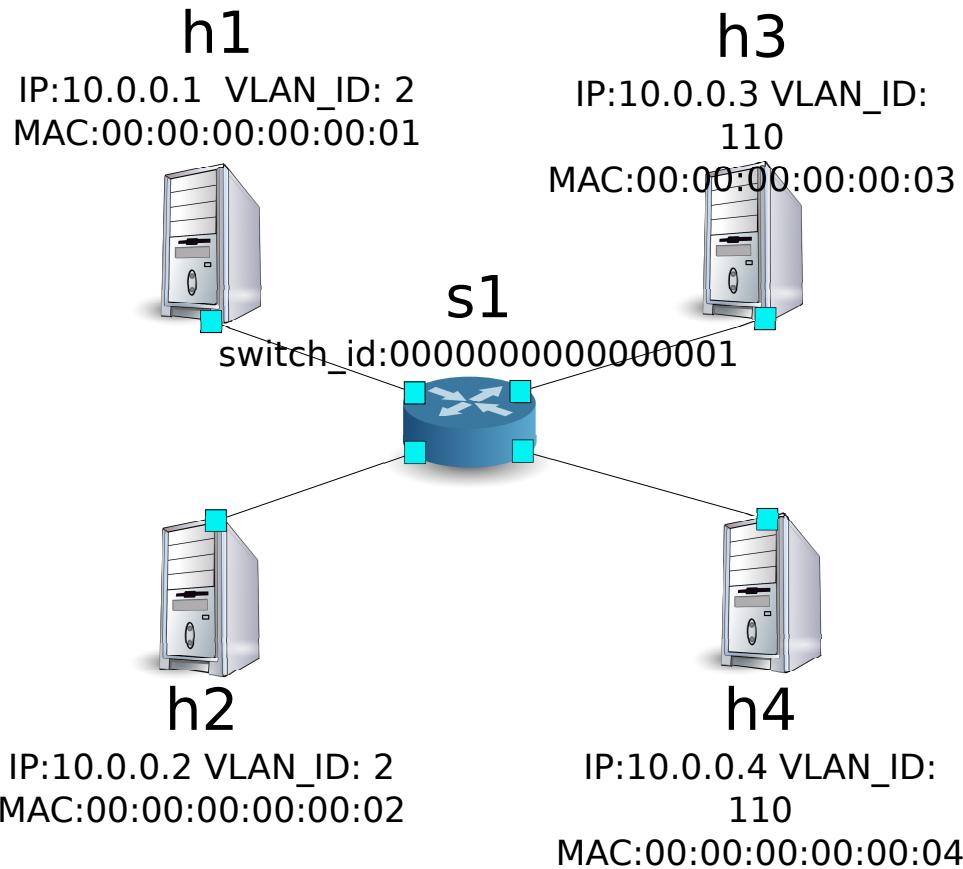
실제로 확인합니다. h2와 h3 사이의 ping (ICMP)을 차단하는 규칙이 삭제 되었기 때문에, ping이 소통 할 수 있게 된 것을 알 수 있습니다.

host: h2:

```
root@ryu-vm:~# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=0.841 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.036 ms
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.026 ms
64 bytes from 10.0.0.3: icmp_req=4 ttl=64 time=0.033 ms
...
```

12.2 멀티 테넌트의 동작 예

이어 VLAN에 의한 임차인 나누기가 이루어지고 있는 다음과 같은 토플로지를 만들고 스위치 s1에 규칙 추가하거나 삭제할 각 호스트 사이의 소통 여부를 확인하는 방법을 소개합니다.



12.2.1 환경 구축

단일 거주자의 예와 마찬가지로 Mininet에 환경을 구축하고 컨트롤러의 xterm 을 또 시작해야합니다. 사용하는 호스트가 하나 증가하고 있는 것에 주의하십시오.

```
ryu@ryu-vm:~$ sudo mn --topo single,4 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 1 switches
s1

*** Starting CLI:
mininet> xterm c0
mininet>
```

이어 각 호스트 인터페이스에 VLAN ID를 설정합니다.

host: h1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1-eth0
root@ryu-vm:~# ip link add link h1-eth0 name h1-eth0.2 type vlan id 2
```

```
root@ryu-vm:~# ip addr add 10.0.0.1/8 dev h1-eth0.2
root@ryu-vm:~# ip link set dev h1-eth0.2 up
```

host: h2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h2-eth0
root@ryu-vm:~# ip link add link h2-eth0 name h2-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 10.0.0.2/8 dev h2-eth0.2
root@ryu-vm:~# ip link set dev h2-eth0.2 up
```

host: h3:

```
root@ryu-vm:~# ip addr del 10.0.0.3/8 dev h3-eth0
root@ryu-vm:~# ip link add link h3-eth0 name h3-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 10.0.0.3/8 dev h3-eth0.110
root@ryu-vm:~# ip link set dev h3-eth0.110 up
```

host: h4:

```
root@ryu-vm:~# ip addr del 10.0.0.4/8 dev h4-eth0
root@ryu-vm:~# ip link add link h4-eth0 name h4-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 10.0.0.4/8 dev h4-eth0.110
root@ryu-vm:~# ip link set dev h4-eth0.110 up
```

또한 사용하는 OpenFlow 버전을 1.3으로 설정합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

마지막으로, 컨트롤러 xterm에서 rest_firewall을 시작합니다.

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_firewall
loading app ryu.app.rest_firewall
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_firewall of RestFirewallAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(13419) wsgi starting up on http://0.0.0.0:8080/
```

Ryu와 스위치 간의 연결에 성공하면 다음 메시지가 표시됩니다.

controller: c0 (root):

```
[FW] [INFO] switch_id=0000000000000001: Join as firewall
```

12.2.2 초기 상태의 변경

firewall을 활성화 (enable)합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/module/enable/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": {
      "result": "success",
      "details": "firewall running."
    }
}
```

```
[  
root@ryu-vm:~# curl http://localhost:8080/firewall/module/status  
[  
 {  
   "status": "enable",  
   "switch_id": "00000000000000000001"  
 }  
]  
]
```

12.2.3 규칙 추가

vlan_id=2에 10.0.0.0/8로 송수신되는 ping (ICMP 패킷)을 허용하는 규칙을 추가 가입합니다. 양방향 규칙을 설정할 필요가 있기 때문에 규칙을 두 추가 있습니다.

(우선순위)	VLAN ID	원본	대상	프로토콜	여부	(규칙 ID)
1	2	10.0.0.0/8	any	ICMP	許可	1
1	2	any	10.0.0.0/8	ICMP	許可	2

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.0/8", "nw_proto": "ICMP"}' http://localhost:8080/firewall/rules/  
[  
 {  
   "switch_id": "00000000000000000001",  
   "command_result": [  
     {  
       "result": "success",  
       "vlan_id": 2,  
       "details": "Rule added. : rule_id=1"  
     }  
   ]  
 }  
]  
  
root@ryu-vm:~# curl -X POST -d '{"nw_dst": "10.0.0.0/8", "nw_proto": "ICMP"}' http://localhost:8080/firewall/rules/  
[  
 {  
   "switch_id": "00000000000000000001",  
   "command_result": [  
     {  
       "result": "success",  
       "vlan_id": 2,  
       "details": "Rule added. : rule_id=2"  
     }  
   ]  
 }  
]
```

12.2.4 규칙 확인

설정된 규칙을 확인합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/00000000000000000001/all  
[  
 {  
   "access_control_list": [  
     {  
       "rules": [  
         {  
           "id": 1,  
           "nw_src": "10.0.0.0/8",  
           "nw_dst": "any",  
           "nw_proto": "ICMP",  
           "action": "allow",  
           "rule_id": 1  
         },  
         {  
           "id": 2,  
           "nw_src": "any",  
           "nw_dst": "10.0.0.0/8",  
           "nw_proto": "ICMP",  
           "action": "allow",  
           "rule_id": 2  
         }  
       ]  
     }  
   ]  
 }
```

```

    {
        "priority": 1,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "dl_vlan": 2,
        "nw_src": "10.0.0.0/8",
        "rule_id": 1,
        "actions": "ALLOW"
    },
    {
        "priority": 1,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "nw_dst": "10.0.0.0/8",
        "dl_vlan": 2,
        "rule_id": 2,
        "actions": "ALLOW"
    }
],
"vlan_id": 2
}
],
"switch_id": "0000000000000001"
}
]
]

```

실제로 확인해 보겠습니다. vlan_id=2의 h1에서, 같은 vlan_id=2의 h2 대해 ping을 실행하면 추가한 규칙과 소통 할 수 있는 것을 알 수 있습니다.

host: h1:

```

root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.893 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.098 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.122 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.047 ms
...

```

vlan_id = 110 사이의 h3와 h4 사이에는 규칙이 등록되어 있지 않기 때문에, ping 패킷 포트는 차단됩니다.

host: h3:

```

root@ryu-vm:~# ping 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
^C
--- 10.0.0.4 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 4999ms

```

패킷이 차단 되었기 때문에 로그가 출력됩니다.

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:04', ethertype=330...
```

이 장에서는 구체적인 예를 들면서 방화벽의 사용 방법을 설명했습니다.

12.3 REST API 목록

이 장에서 소개 한 rest_firewall의 REST API를 나열합니다.

12.3.1 모든 스위치의 사용 가능 상태의 추적

메소드	GET
URL	/firewall/module/status

12.3.2 각 스위치의 사용 가능 상태 변경

메소드	PUT
URL	/firewall/module/{op}/{switch} -op: [“enable” “disable”] -switch: [“all” 스위치ID]
주의	각 스위치의 초기 상태는 “disable”로 되어 있습니다.

12.3.3 모든 규칙 가져오기

메소드	GET
URL	/firewall/rules/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
주의	VLAN ID의 지정은 선택 사항입니다.

12.3.4 규칙 추가

메소드	POST
URL	/firewall/rules/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	priority:[0 - 65535] in_port:[0 - 65535] dl_src:”<XX:XX:XX:XX:XX:XX>” dl_dst:”<XX:XX:XX:XX:XX:XX>” dl_type:[“ARP” “IPv4”] nw_src:”<XXX.XXX.XXX.XXX/XX>” nw_dst:”<XXX.XXX.XXX.XXX/XX>” nw_proto:[“TCP” “UDP” “ICMP”] tp_src:[0 - 65535] tp_dst:[0 - 65535] actions: [“ALLOW” “DENY”]
주의	등록에 성공하면 규칙 ID가 생성되어 응답에 포함됩니다. VLAN ID의 지정은 선택 사항입니다.

12.3.5 규칙 삭제

메소드	DELETE
URL	/firewall/rules/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	rule_id:[“all” 1 - ...]
주의	VLAN ID의 지정은 선택 사항입니다.

12.3.6 모든 스위치 로깅 상태 가져 오기

메소드	GET
URL	/firewall/log/status

12.3.7 각 스위치의 로깅 상태 변경

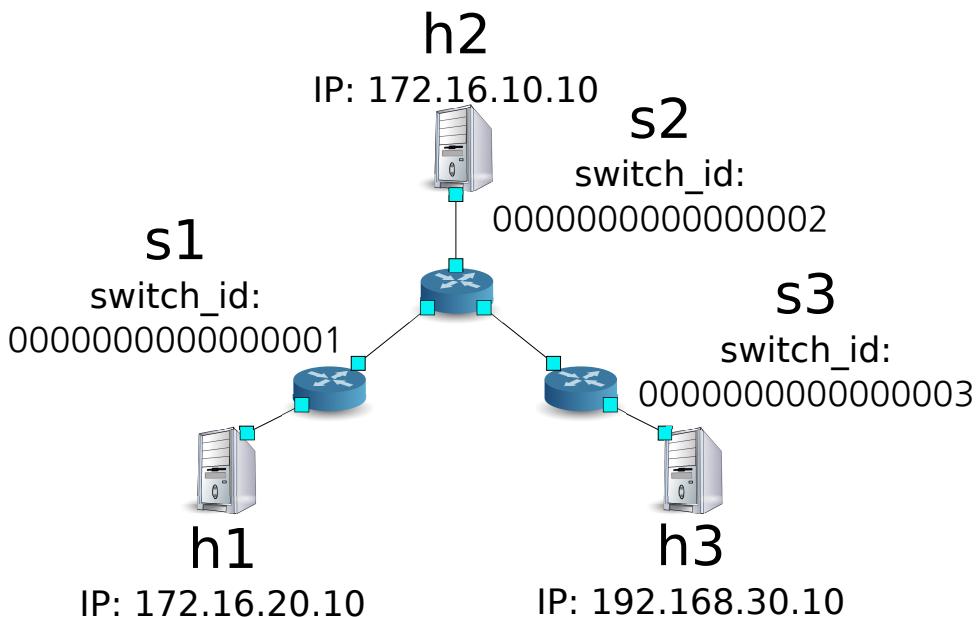
메소드	PUT
URL	/firewall/log/{op}/{switch} -op: [“enable” “disable”] -switch: [“all” 스위치ID]
주의	각 스위치의 초기 상태는 “enable”로되어 있습니다.

라우터

이 장에서는 REST에서 설정이 가능한 라우터를 사용하는 방법에 대해 설명합니다.

13.1 단일 거주자의 동작 예

다음과 같은 토플로지를 만들고 각 스위치(라우터)에 주소와 경로를 추가하거나 삭제할 각 호스트 간의 소통 가능 여부를 확인하는 방법을 소개합니다.



13.1.1 환경 구축

우선 Mininet에 환경을 구축합니다. `mn` 명령의 매개 변수는 다음과 같습니다.

매개변수	값	설명
topo	linear,3	3 개의 스위치가 직렬로 연결되는 토플로지
mac	없음	자동으로 호스트의 MAC 주소를 설정한다
switch	ovsk	Open vSwitch를 사용
controller	remote	OpenFlow 컨트롤러는 외부의 것을 이용하기
x	없음	xterm을 시작

실행 예는 다음과 같습니다.

```
ryu@ryu-vm:~$ sudo mn --topo linear,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1 h2 h3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 3 switches
s1 s2 s3

*** Starting CLI:
mininet>
```

또한 컨트롤러의 xterm을 또 시작해야합니다.

```
mininet> xterm c0
mininet>
```

이어 각 라우터에서 사용하는 OpenFlow 버전을 1.3으로 설정합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

switch: s2 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

switch: s3 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s3 protocols=OpenFlow13
```

그린 다음 각 호스트에서 자동으로 할당 된 IP 주소를 삭제하고 새로운 IP 주소를 설정합니다.

host: h1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1-eth0
root@ryu-vm:~# ip addr add 172.16.20.10/24 dev h1-eth0
```

host: h2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h2-eth0
root@ryu-vm:~# ip addr add 172.16.10.10/24 dev h2-eth0
```

host: h3:

```
root@ryu-vm:~# ip addr del 10.0.0.3/8 dev h3-eth0
root@ryu-vm:~# ip addr add 192.168.30.10/24 dev h3-eth0
```

마지막으로, 컨트롤러 xterm에서 rest_router을 시작합니다.

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_router
loading app ryu.app.rest_router
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
```

```
instantiating app ryu.app.rest_router of RestRouterAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(2212) wsgi starting up on http://0.0.0.0:8080/
```

Ryu와 라우터 간의 연결에 성공하면 다음 메시지가 표시됩니다.

controller: c0 (root):

```
[RT] [INFO] switch_id=0000000000000003: Set SW config for TTL error packet in.
[RT] [INFO] switch_id=0000000000000003: Set ARP handling (packet in) flow [cookie=0x0]
[RT] [INFO] switch_id=0000000000000003: Set L2 switching (normal) flow [cookie=0x0]
[RT] [INFO] switch_id=0000000000000003: Set default route (drop) flow [cookie=0x0]
[RT] [INFO] switch_id=0000000000000003: Start cyclic routing table update.
[RT] [INFO] switch_id=0000000000000003: Join as router.
...
```

위 로그 라우터 3 대분 표시되면 준비 완료입니다.

13.1.2 주소 설정

각 라우터에 주소를 설정합니다.

먼저 라우터 s1 주소 「172.16.20.1/24」와 「172.16.30.30/24」를 설정합니다.

주석: 이후의 설명에서 사용하는 REST API의 자세한 내용은 장 끝부분의 「REST API 목록」을 참조하십시오.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.20.1/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Add address [address_id=1]"
      }
    ]
  }
]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.30.30/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Add address [address_id=2]"
      }
    ]
  }
]
```

주석: REST 명령의 실행 결과는 보기 쉽도록 형성하고 있습니다.

그런 다음 라우터 s2에 주소 「172.16.10.1/24」 「172.16.30.1/24」 「192.168.10.1/24」을 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/0000000000000002
[
  {
    "switch_id": "0000000000000002",
    "command_result": [
      {
        "result": "success",
        "details": "Add address [address_id=3]"
      }
    ]
  }
]
```

```

        "switch_id": "0000000000000002",
        "command_result": [
            {
                "result": "success",
                "details": "Add address [address_id=1]"
            }
        ]
    }
]
]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.30.1/24"}' http://localhost:8080/router/0000000000000002
[
{
    "switch_id": "0000000000000002",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=2]"
        }
    ]
}
]
]

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.10.1/24"}' http://localhost:8080/router/0000000000000002
[
{
    "switch_id": "0000000000000002",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=3]"
        }
    ]
}
]
]
]

또한 라우터 s3에 주소 「192.168.30.1/24」와 「192.168.10.20/24」을 설정 합니다.
```

Node: c0 (root):

```

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost:8080/router/0000000000000003
[
{
    "switch_id": "0000000000000003",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=1]"
        }
    ]
}
]
]

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.10.20/24"}' http://localhost:8080/router/0000000000000003
[
{
    "switch_id": "0000000000000003",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=2]"
        }
    ]
}
]
]
```

```

        }
    ]
}
```

라우터에 IP 주소를 할당 할 수 있기 때문에 각 호스트에 기본 게이트웨이로 등록합니다.

host: h1:

```
root@ryu-vm:~# ip route add default via 172.16.20.1
```

host: h2:

```
root@ryu-vm:~# ip route add default via 172.16.10.1
```

host: h3:

```
root@ryu-vm:~# ip route add default via 192.168.30.1
```

13.1.3 기본 경로 설정

각 라우터에 기본 경로를 설정합니다.

먼저 라우터 s1의 기본 경로로 라우터 s2를 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.1"}' http://localhost:8080/router/00000000000000000000000000000000
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Add route [route_id=1]"}]}
```

라우터 s2의 기본 경로는 라우터 s1을 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.30"}' http://localhost:8080/router/00000000000000000000000000000001
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "Add route [route_id=1]"}]}
```

라우터 s3의 기본 경로는 라우터 s2를 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "192.168.10.1"}' http://localhost:8080/router/00000000000000000000000000000002
[{"switch_id": "0000000000000003", "command_result": [{"result": "success", "details": "Add route [route_id=1]"}]}
```

```

        "result": "success",
        "details": "Add route [route_id=1]"
    }
]
}
]

```

13.1.4 정적 경로 설정

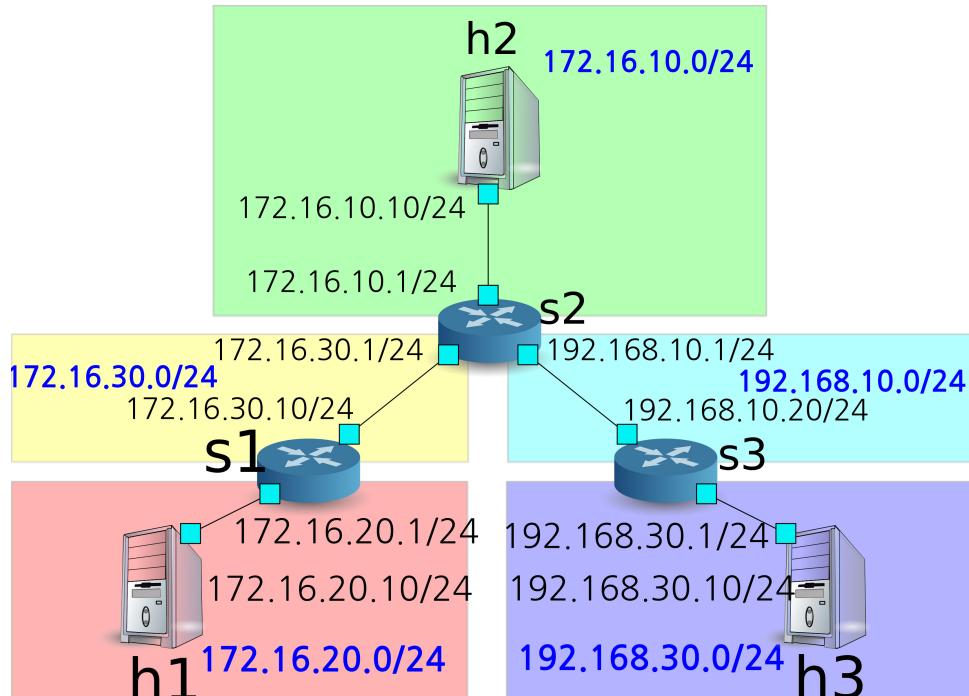
라우터 s2에 대해 라우터 s3 부하의 호스트 (192.168.30.0/24)에 고정 경로를 설정합니다.

Node: c0 (root):

```

root@ryu-vm:~# curl -X POST -d '{"destination": "192.168.30.0/24", "gateway": "192.168.10.20"}' http://127.0.0.1:8080/router/0000000000000000
[
{
    "switch_id": "0000000000000002",
    "command_result": [
        {
            "result": "success",
            "details": "Add route [route_id=2]"
        }
    ]
}
]
```

주소 및 경로 설정 상태는 다음과 같습니다.



13.1.5 설정 내용 확인

각 라우터에 설정된 내용을 확인합니다.

Node: c0 (root):

```

root@ryu-vm:~# curl http://localhost:8080/router/0000000000000001
[
{

```

```

"internal_network": [
    {
        "route": [
            {
                "route_id": 1,
                "destination": "0.0.0.0/0",
                "gateway": "172.16.30.1"
            }
        ],
        "address": [
            {
                "address_id": 1,
                "address": "172.16.20.1/24"
            },
            {
                "address_id": 2,
                "address": "172.16.30.30/24"
            }
        ]
    },
    "switch_id": "00000000000000000001"
}
]
]

root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000002
[
{
    "internal_network": [
        {
            "route": [
                {
                    "route_id": 1,
                    "destination": "0.0.0.0/0",
                    "gateway": "172.16.30.30"
                },
                {
                    "route_id": 2,
                    "destination": "192.168.30.0/24",
                    "gateway": "192.168.10.20"
                }
            ],
            "address": [
                {
                    "address_id": 2,
                    "address": "172.16.30.1/24"
                },
                {
                    "address_id": 3,
                    "address": "192.168.10.1/24"
                },
                {
                    "address_id": 1,
                    "address": "172.16.10.1/24"
                }
            ]
        }
    ],
    "switch_id": "00000000000000000002"
}
]
]

root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000003

```

```
[
  {
    "internal_network": [
      {
        "route": [
          {
            "route_id": 1,
            "destination": "0.0.0.0/0",
            "gateway": "192.168.10.1"
          }
        ],
        "address": [
          {
            "address_id": 1,
            "address": "192.168.30.1/24"
          },
          {
            "address_id": 2,
            "address": "192.168.10.20/24"
          }
        ]
      }
    ],
    "switch_id": "00000000000000000003"
  }
]
```

이 상태에서 ping에 의한 소통을 확인하여보십시오. 먼저 h2에서 h3에 ping을 수행합니다 입니다. 성공적으로 소통 할 수 있는 것을 확인할 수 있습니다.

host: h2:

```
root@ryu-vm:~# ping 192.168.30.10
PING 192.168.30.10 (192.168.30.10) 56(84) bytes of data.
64 bytes from 192.168.30.10: icmp_req=1 ttl=62 time=48.8 ms
64 bytes from 192.168.30.10: icmp_req=2 ttl=62 time=0.402 ms
64 bytes from 192.168.30.10: icmp_req=3 ttl=62 time=0.089 ms
64 bytes from 192.168.30.10: icmp_req=4 ttl=62 time=0.065 ms
...
...
```

또한 h2에서 h1로 ping을 실행합니다. 이쪽도 제대로 소통 할 수 있는지 확인 할 수 있습니다.

host: h2:

```
root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
64 bytes from 172.16.20.10: icmp_req=1 ttl=62 time=43.2 ms
64 bytes from 172.16.20.10: icmp_req=2 ttl=62 time=0.306 ms
64 bytes from 172.16.20.10: icmp_req=3 ttl=62 time=0.057 ms
64 bytes from 172.16.20.10: icmp_req=4 ttl=62 time=0.048 ms
...
...
```

13.1.6 정적 경로 삭제

라우터 s2에 설정 한 라우터 s3에 정적 경로를 제거합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"route_id": "2"}' http://localhost:8080/router/00000000000000000000000000000002
[
  {
    "switch_id": "00000000000000000000000000000002",
    "command_result": [
      ...
    ]
  }
]
```

```

        {
            "result": "success",
            "details": "Delete route [route_id=2]"
        }
    ]
}
]
]
```

라우터 s2에 설정된 정보를 확인하여보십시오. 라우터 s3에 고정 경로가 삭제 된 것을 알 수 있습니다.

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000002
[
  {
    "internal_network": [
      {
        "route": [
          {
            "route_id": 1,
            "destination": "0.0.0.0/0",
            "gateway": "172.16.30.30"
          }
        ],
        "address": [
          {
            "address_id": 2,
            "address": "172.16.30.1/24"
          },
          {
            "address_id": 3,
            "address": "192.168.10.1/24"
          },
          {
            "address_id": 1,
            "address": "172.16.10.1/24"
          }
        ]
      }
    ],
    "switch_id": "0000000000000002"
  }
]
```

이 상태에서 ping에 의한 소통을 확인하여보십시오. h2에서 h3까지는 노선 정보가 없어 때문에 소통 할 수 없는 것을 알 수 있습니다.

host: h2:

```
root@ryu-vm:~# ping 192.168.30.10
PING 192.168.30.10 (192.168.30.10) 56(84) bytes of data.
^C
--- 192.168.30.10 ping statistics ---
12 packets transmitted, 0 received, 100% packet loss, time 11088ms
```

13.1.7 주소 삭제

라우터 s1에 설정 한 주소 「172.16.20.1/24」를 삭제합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"address_id": "1"}' http://localhost:8080/router/00000000000000000001
[
```

```
{
  "switch_id": "0000000000000001",
  "command_result": [
    {
      "result": "success",
      "details": "Delete address [address_id=1]"
    }
  ]
}
```

라우터 s1에 설정된 정보를 확인하여보십시오. 라우터 s1에 설정된 IP 주소 중 「172.16.20.1/24」가 삭제 된 것을 알 수 있습니다.

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000001
[
  {
    "internal_network": [
      {
        "route": [
          {
            "route_id": 1,
            "destination": "0.0.0.0/0",
            "gateway": "172.16.30.1"
          }
        ],
        "address": [
          {
            "address_id": 2,
            "address": "172.16.30.30/24"
          }
        ]
      }
    ],
    "switch_id": "0000000000000001"
  }
]
```

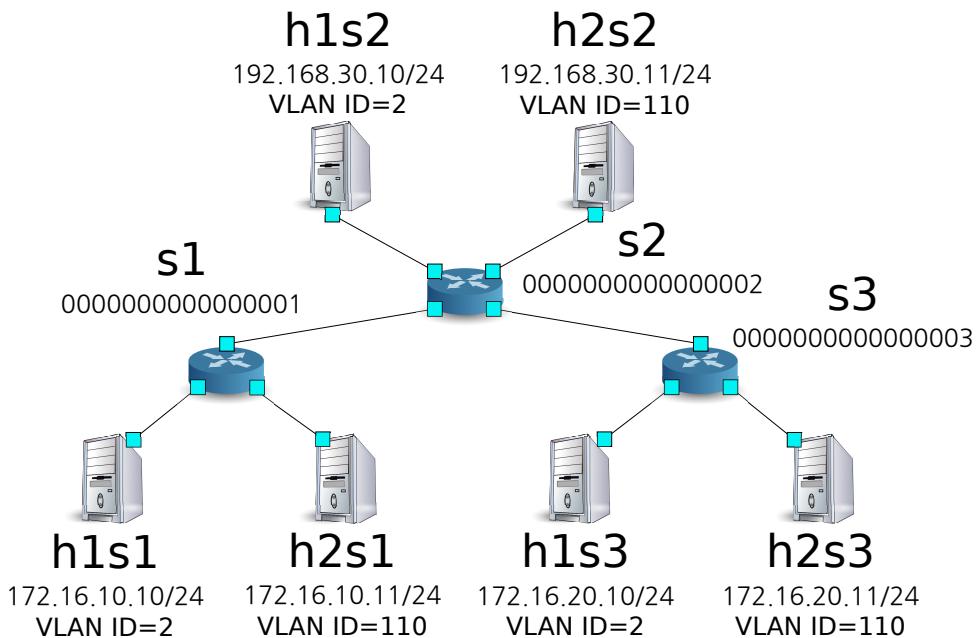
이 상태에서 ping에 의한 소통을 확인하여보십시오. h2에서 h1으로는 h1 속한 하위 인터넷에 대한 정보를 라우터 s1에서 삭제 되었기 때문에, 소통 할 수 없는 것을 알 수 없습니다.

host: h2:

```
root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
^C
--- 172.16.20.10 ping statistics ---
19 packets transmitted, 0 received, 100% packet loss, time 18004ms
```

13.2 멀티 테넌트의 동작 예

이어 VLAN에 의한 임차인 나누기가 이루어지고있는 다음과 같은 토플로지를 만들고 각 스위치 (라우터)에 주소와 경로를 추가하거나 삭제할 각 호스트 간의 소통 여부를 확인하는 방법을 소개합니다.



13.2.1 환경 구축

우선 Mininet에 환경을 구축합니다. mn 명령의 매개 변수는 다음과 같이입니다.

매개변수	값	설명
topo	linear,3,2	3 개의 스위치가 직렬로 연결되는 토플로지 (각 스위치에 2 개의 호스트가 연결되는)
mac	없음	자동으로 호스트의 MAC 주소를 설정
switch	ovsk	Open vSwitch를 사용함
controller	remote	OpenFlow 컨트롤러는 외부의 것을 이용하기
x	없음	xterm을 시작

실행 예는 다음과 같습니다.

```
ryu@ryu-vm:~$ sudo mn --topo linear,3,2 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1s1 h1s2 h1s3 h2s1 h2s2 h2s3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1s1, s1) (h1s2, s2) (h1s3, s3) (h2s1, s1) (h2s2, s2) (h2s3, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1s1 h1s2 h1s3 h2s1 h2s2 h2s3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 3 switches
s1 s2 s3
*** Starting CLI:
mininet>
```

또한 컨트롤러의 xterm을 또 시작해야합니다.

```
mininet> xterm c0
mininet>
```

이어 각 라우터에서 사용하는 OpenFlow 버전을 1.3으로 설정합니다.

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

switch: s2 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

switch: s3 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s3 protocols=OpenFlow13
```

그린 다음 각 호스트 인터페이스에 VLAN ID를 설정하고 새로운 IP 주소를 설정 합니다.

host: h1s1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1s1-eth0
root@ryu-vm:~# ip link add link h1s1-eth0 name h1s1-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 172.16.10.10/24 dev h1s1-eth0.2
root@ryu-vm:~# ip link set dev h1s1-eth0.2 up
```

host: h2s1:

```
root@ryu-vm:~# ip addr del 10.0.0.4/8 dev h2s1-eth0
root@ryu-vm:~# ip link add link h2s1-eth0 name h2s1-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 172.16.10.11/24 dev h2s1-eth0.110
root@ryu-vm:~# ip link set dev h2s1-eth0.110 up
```

host: h1s2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h1s2-eth0
root@ryu-vm:~# ip link add link h1s2-eth0 name h1s2-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 192.168.30.10/24 dev h1s2-eth0.2
root@ryu-vm:~# ip link set dev h1s2-eth0.2 up
```

host: h2s2:

```
root@ryu-vm:~# ip addr del 10.0.0.5/8 dev h2s2-eth0
root@ryu-vm:~# ip link add link h2s2-eth0 name h2s2-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 192.168.30.11/24 dev h2s2-eth0.110
root@ryu-vm:~# ip link set dev h2s2-eth0.110 up
```

host: h1s3:

```
root@ryu-vm:~# ip addr del 10.0.0.3/8 dev h1s3-eth0
root@ryu-vm:~# ip link add link h1s3-eth0 name h1s3-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 172.16.20.10/24 dev h1s3-eth0.2
root@ryu-vm:~# ip link set dev h1s3-eth0.2 up
```

host: h2s3:

```
root@ryu-vm:~# ip addr del 10.0.0.6/8 dev h2s3-eth0
root@ryu-vm:~# ip link add link h2s3-eth0 name h2s3-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 172.16.20.11/24 dev h2s3-eth0.110
root@ryu-vm:~# ip link set dev h2s3-eth0.110 up
```

마지막으로, 컨트롤러 xterm에서 rest_router을 시작합니다.

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_router
loading app ryu.app.rest_router
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_router of RestRouterAPI
```

```
instantiating app ryu.controller.ofp_handler of OFPHandler
(2447) wsgi starting up on http://0.0.0.0:8080/
```

Ryu와 라우터 간의 연결에 성공하면 다음 메시지가 표시됩니다.

controller: c0 (root):

```
[RT] [INFO] switch_id=0000000000000003: Set SW config for TTL error packet in.
[RT] [INFO] switch_id=0000000000000003: Set ARP handling (packet in) flow [cookie=0x0]
[RT] [INFO] switch_id=0000000000000003: Set L2 switching (normal) flow [cookie=0x0]
[RT] [INFO] switch_id=0000000000000003: Set default route (drop) flow [cookie=0x0]
[RT] [INFO] switch_id=0000000000000003: Start cyclic routing table update.
[RT] [INFO] switch_id=0000000000000003: Join as router.
...
```

위 로그 라우터 3 대분 표시되면 준비 완료입니다.

13.2.2 주소 설정

각 라우터에 주소를 설정합니다.

먼저 라우터 s1 주소 「172.16.20.1/24」와 「10.10.10.1/24」을 설정합니다. 입니다. 각 VLAN ID마다 설정해야 합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 2,
        "details": "Add address [address_id=1]"
      }
    ]
  }
]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.1/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 2,
        "details": "Add address [address_id=2]"
      }
    ]
  }
]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 110,
        "details": "Add address [address_id=1]"
      }
    ]
  }
]
```

```

        }
    ]
}
]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.1/24"}' http://localhost:8080/router/00000000000000000000000000000001
[
{
  "switch_id": "00000000000000000000000000000001",
  "command_result": [
    {
      "result": "success",
      "vlan_id": 110,
      "details": "Add address [address_id=2]"
    }
  ]
}
]
]
```

그런 다음 라우터 s2에 주소 「192.168.30.1/24」와 「10.10.10.2/24」을 설정 합니다.

Node: c0 (root):

```

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost:8080/router/00000000000000000000000000000002
[
{
  "switch_id": "00000000000000000000000000000002",
  "command_result": [
    {
      "result": "success",
      "vlan_id": 2,
      "details": "Add address [address_id=1]"
    }
  ]
}
]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.2/24"}' http://localhost:8080/router/00000000000000000000000000000002
[
{
  "switch_id": "00000000000000000000000000000002",
  "command_result": [
    {
      "result": "success",
      "vlan_id": 2,
      "details": "Add address [address_id=2]"
    }
  ]
}
]

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost:8080/router/00000000000000000000000000000002
[
{
  "switch_id": "00000000000000000000000000000002",
  "command_result": [
    {
      "result": "success",
      "vlan_id": 110,
      "details": "Add address [address_id=1]"
    }
  ]
}
]
```

```
[  
root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.2/24"}' http://localhost:8080/router/00000000000000000000000000000000  
[  
{  
    "switch_id": "00000000000000000000000000000002",  
    "command_result": [  
        {  
            "result": "success",  
            "vlan_id": 110,  
            "details": "Add address [address_id=2]"  
        }  
    ]  
}  
]  
]
```

또한 라우터 s3에 주소 「172.16.20.1/24」와 「10.10.10.3/24」을 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.20.1/24"}' http://localhost:8080/router/00000000000000000000000000000003  
[  
{  
    "switch_id": "00000000000000000000000000000003",  
    "command_result": [  
        {  
            "result": "success",  
            "vlan_id": 2,  
            "details": "Add address [address_id=1]"  
        }  
    ]  
}  
]  
]  
  
root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.3/24"}' http://localhost:8080/router/00000000000000000000000000000003  
[  
{  
    "switch_id": "00000000000000000000000000000003",  
    "command_result": [  
        {  
            "result": "success",  
            "vlan_id": 2,  
            "details": "Add address [address_id=2]"  
        }  
    ]  
}  
]  
]  
  
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.20.1/24"}' http://localhost:8080/router/00000000000000000000000000000003  
[  
{  
    "switch_id": "00000000000000000000000000000003",  
    "command_result": [  
        {  
            "result": "success",  
            "vlan_id": 110,  
            "details": "Add address [address_id=1]"  
        }  
    ]  
}  
]  
]  
  
root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.3/24"}' http://localhost:8080/router/00000000000000000000000000000003
```

```
[
  {
    "switch_id": "0000000000000003",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 110,
        "details": "Add address [address_id=2]"
      }
    ]
  }
]
```

라우터에 IP 주소를 할당 할 수 있기 때문에 각 호스트에 기본 게이트웨이로 등록합니다.

host: h1s1:

```
root@ryu-vm:~# ip route add default via 172.16.10.1
```

host: h2s1:

```
root@ryu-vm:~# ip route add default via 172.16.10.1
```

host: h1s2:

```
root@ryu-vm:~# ip route add default via 192.168.30.1
```

host: h2s2:

```
root@ryu-vm:~# ip route add default via 192.168.30.1
```

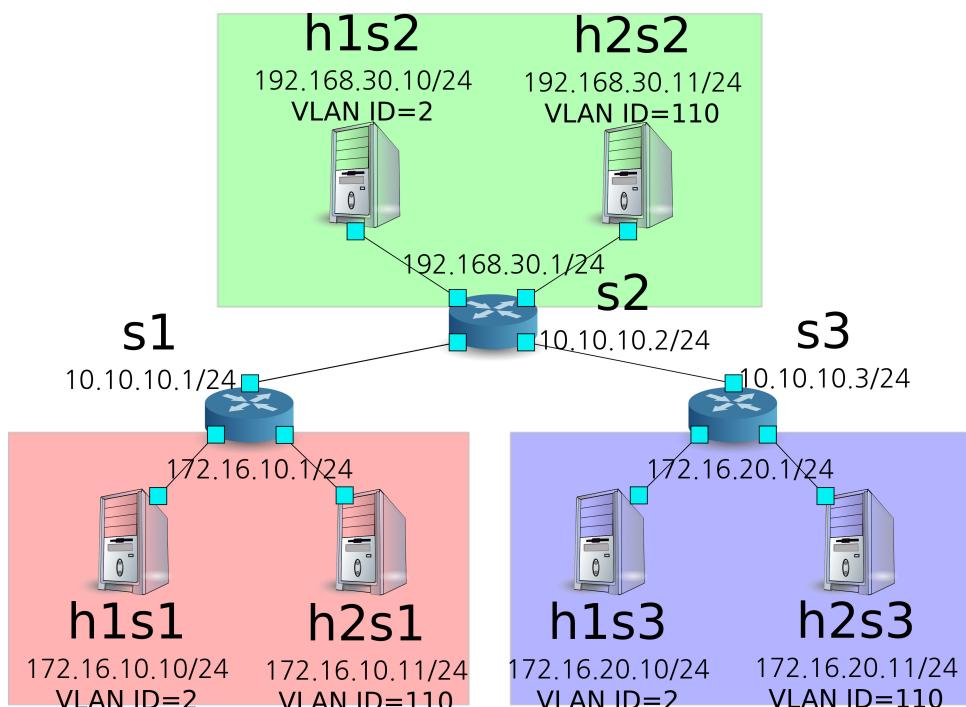
host: h1s3:

```
root@ryu-vm:~# ip route add default via 172.16.20.1
```

host: h2s3:

```
root@ryu-vm:~# ip route add default via 172.16.20.1
```

설정된 주소는 다음과 같습니다.



13.2.3 기본 경로 및 정적 경로 설정

각 라우터에 기본 경로 및 정적 경로를 설정합니다.

먼저 라우터 s1의 기본 경로로 라우터 s2를 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/00000000000000000000000000000001
[{"switch_id": "00000000000000000000000000000001", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=1]"}]}]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/00000000000000000000000000000001
[{"switch_id": "00000000000000000000000000000001", "command_result": [{"result": "success", "vlan_id": 110, "details": "Add route [route_id=1]"}]}]
```

라우터 s2의 기본 경로는 라우터 s1을 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/router/00000000000000000000000000000002
[{"switch_id": "00000000000000000000000000000002", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=1]"}]}]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/router/00000000000000000000000000000002
[{"switch_id": "00000000000000000000000000000002", "command_result": [{"result": "success", "vlan_id": 110, "details": "Add route [route_id=1]"}]}]
```

```

        }
    ]
}
```

라우터 s3의 기본 경로는 라우터 s2를 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/00000000000000000000000000000003
[{"switch_id": "000000000000000000000003", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=1]"}]}
]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/000000000000000000000003
[{"switch_id": "000000000000000000000003", "command_result": [{"result": "success", "vlan_id": 110, "details": "Add route [route_id=1]"}]}
]
```

이어 라우터 s2에 대해 라우터 s3 부하의 호스트 (172.16.20.0/24)의 정적 경로를 설정합니다. vlan_id=2 인 경우에만 설정합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"destination": "172.16.20.0/24", "gateway": "10.10.10.3"}' http://localhost:8080/router/000000000000000000000002
[{"switch_id": "000000000000000000000002", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=2]"}]}
]
```

13.2.4 설정 내용 확인

각 라우터에 설정된 내용을 확인합니다.

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/all/all
[{"switch_id": "000000000000000000000003", "command_result": [{"result": "success", "details": "Get routes [route_id=1]"}], "switch_id": "000000000000000000000002", "command_result": [{"result": "success", "details": "Get routes [route_id=2]"}]}
```

```

"internal_network": [
    {},
    {
        "route": [
            {
                "route_id": 1,
                "destination": "0.0.0.0/0",
                "gateway": "10.10.10.2"
            }
        ],
        "vlan_id": 2,
        "address": [
            {
                "address_id": 2,
                "address": "10.10.10.1/24"
            },
            {
                "address_id": 1,
                "address": "172.16.10.1/24"
            }
        ]
    },
    {
        "route": [
            {
                "route_id": 1,
                "destination": "0.0.0.0/0",
                "gateway": "10.10.10.2"
            }
        ],
        "vlan_id": 110,
        "address": [
            {
                "address_id": 2,
                "address": "10.10.10.1/24"
            },
            {
                "address_id": 1,
                "address": "172.16.10.1/24"
            }
        ]
    }
],
"switch_id": "0000000000000001"
},
{
    "internal_network": [
        {},
        {
            "route": [
                {
                    "route_id": 2,
                    "destination": "172.16.20.0/24",
                    "gateway": "10.10.10.3"
                },
                {
                    "route_id": 1,
                    "destination": "0.0.0.0/0",
                    "gateway": "10.10.10.1"
                }
            ],
            "vlan_id": 2,
            "address": [

```

```
{  
    "address_id": 2,  
    "address": "10.10.10.2/24"  
,  
{  
    "address_id": 1,  
    "address": "192.168.30.1/24"  
}  
]  
},  
{  
    "route": [  
        {  
            "route_id": 1,  
            "destination": "0.0.0.0/0",  
            "gateway": "10.10.10.1"  
        }  
],  
    "vlan_id": 110,  
    "address": [  
        {  
            "address_id": 2,  
            "address": "10.10.10.2/24"  
        },  
        {  
            "address_id": 1,  
            "address": "192.168.30.1/24"  
        }  
    ]  
},  
    "switch_id": "0000000000000002"  
},  
{  
    "internal_network": [  
        {},  
        {  
            "route": [  
                {  
                    "route_id": 1,  
                    "destination": "0.0.0.0/0",  
                    "gateway": "10.10.10.2"  
                }  
            ],  
            "vlan_id": 2,  
            "address": [  
                {  
                    "address_id": 1,  
                    "address": "172.16.20.1/24"  
                },  
                {  
                    "address_id": 2,  
                    "address": "10.10.10.3/24"  
                }  
            ]  
        },  
        {  
            "route": [  
                {  
                    "route_id": 1,  
                    "destination": "0.0.0.0/0",  
                    "gateway": "10.10.10.2"  
                }  
            ]  
        }  
    ]  
},  
}
```

```

        ],
        "vlan_id": 110,
        "address": [
            {
                "address_id": 1,
                "address": "172.16.20.1/24"
            },
            {
                "address_id": 2,
                "address": "10.10.10.3/24"
            }
        ]
    ],
    "switch_id": "0000000000000003"
}
]

```

각 라우터의 설정 내용을 표로 하면 다음과 같습니다.

라우터	VLAN ID	IP 주소	기본 경로	고정 경로
s1	2	172.16.10.1/24, 10.10.10.1/24	10.10.10.2(s2)	
s1	110	172.16.10.1/24, 10.10.10.1/24	10.10.10.2(s2)	
s2	2	192.168.30.1/24, 10.10.10.2/24	10.10.10.1(s1) 대상:172.16.20.0/24, 게이트웨이:10.10.10.3(s3)	
s2	110	192.168.30.1/24, 10.10.10.2/24	10.10.10.1(s1)	
s3	2	172.16.20.1/24, 10.10.10.3/24	10.10.10.2(s2)	
s3	110	172.16.20.1/24, 10.10.10.3/24	10.10.10.2(s2)	

h1s1에서 h1s3 대해 ping을 시도합니다. 같은 vlan_id=2의 호스트끼리이며, 라우터 s2에 s3에게 고정 경로가 설정되어 있기 때문에 소통이 가능합니다.

host: h1s1:

```

root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
64 bytes from 172.16.20.10: icmp_req=1 ttl=61 time=45.9 ms
64 bytes from 172.16.20.10: icmp_req=2 ttl=61 time=0.257 ms
64 bytes from 172.16.20.10: icmp_req=3 ttl=61 time=0.059 ms
64 bytes from 172.16.20.10: icmp_req=4 ttl=61 time=0.182 ms

```

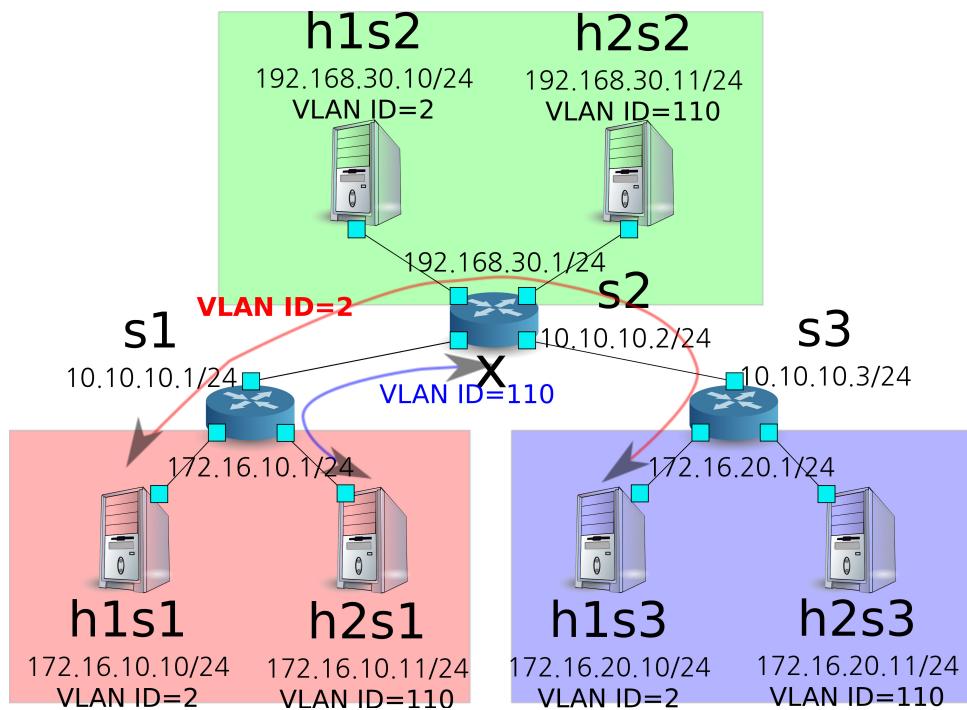
h2s1에서 h2s3 대해 ping을 시도합니다. 같은 vlan_id=110 호스트끼리이지만, 라우터 s2에 s3에게 고정 경로가 설정되어 있지 않기 때문에 소통이 불가능합니다.

host: h2s1:

```

root@ryu-vm:~# ping 172.16.20.11
PING 172.16.20.11 (172.16.20.11) 56(84) bytes of data.
^C
--- 172.16.20.11 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7009ms

```



이 장에서는 구체적인 예를 들면서 라우터의 사용 방법을 설명했습니다.

13.3 REST API 목록

이 장에서 소개 한 rest_router의 REST API를 나열합니다.

13.3.1 설정 가져오기

메소드	GET
URL	/router/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
주의	VLAN ID의 지정은 선택 사항입니다.

13.3.2 주소 설정

메소드	POST
URL	/router/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	address:"<xxx.xxx.xxx.xxx/xx>"
주의	주소 설정은 루트 설정 전에 수행해야합니다. VLAN ID의 지정은 선택 사항입니다.

13.3.3 정적 경로 설정

메소드	POST
URL	/router/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	destination:<xxx.xxx.xxx.xxx/xx> gateway:<xxx.xxx.xxx.xxx>
주의	VLAN ID의 지정은 선택 사항입니다.

13.3.4 디폴트 경로 설정

메소드	POST
URL	/router/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	gateway:<xxx.xxx.xxx.xxx>
주의	VLAN ID의 지정은 선택 사항입니다.

13.3.5 주소 삭제

메소드	DELETE
URL	/router/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	address_id:[1 - ...]
주의	VLAN ID의 지정은 선택 사항입니다.

13.3.6 루트 삭제

메소드	DELETE
URL	/router/{switch}[/{vlan}] -switch: [“all” 스위치ID] -vlan: [“all” VLAN ID]
데이터	route_id:[1 - ...]
주의	VLAN ID의 지정은 선택 사항입니다.

OpenFlow 스위치 테스트 도구

이 장에서는 OpenFlow 스위치 OpenFlow 사양 준수의 정도를 검증하는 테스트 도구의 사용 방법을 설명합니다.

14.1 테스트 도구의 개요

본 도구는 테스트 패턴에 따라 시험 할 OpenFlow 스위치 대해 흐름 항목 및 측정기 항목의 등록 / 패킷 인가를 실시하고, OpenFlow 스위치의 패킷 재 작성 및 전송 (또는 삭제)의 처리 결과와 테스트 패턴 파일에 포함 된 「기대하는 처리 결과」의 비교를 실시하는 것으로, OpenFlow 스위치 OpenFlow 사양에의 대응 상황을 확인하는 테스트 도구입니다.

도구는 OpenFlow 버전 1.3 FlowMod 메시지 및 MeterMod 메시지 시험에 대응하고 있습니다.

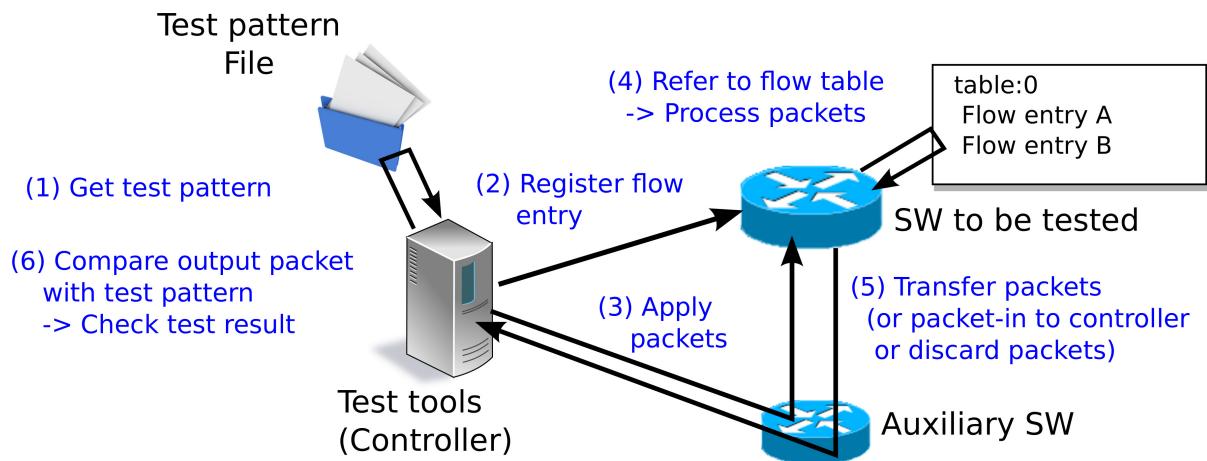
시험 대상 메시지	대응 매개변수
OpenFlow1.3 FlowMod 메시지	match (IN_PHY_PORT 제외) actions (SET_QUEUE, GROUP 제외)
OpenFlow1.3 MeterMod 메시지	모두

인가하는 패킷의 생성과 패킷 재 작성 결과의 확인 등에 「[패킷 라이브러리](#)」를 사용하고 있습니다.

14.1.1 동작 개요

시험 실행 이미지

테스트 도구를 실행했을 때의 동작 이미지를 보여줍니다. 테스트 패턴 파일에는 「등록 흐름 항목 또는 미터 항목」「인가 패킷」「기대하는 처리 결과」가 설명됩니다. 또한 도구 실행을 위한 환경 설정 내용은 뒤에 서술 ([도구 실행 환경 참조](#))합니다.



시험 결과의 출력 이미지

지정된 테스트 패턴 테스트 항목을 순서대로 수행하고 시험 결과 (OK/ERROR)를 출력합니다. 시험 결과가 ERROR의 경우 오류 정보를 함께 출력합니다. 또한 시험 전체의 OK/ERROR 수 및 발생한 ERROR 내역도 출력합니다.

--- Test start ---

```

match: 29_ICMPV6_TYPE
    ethernet/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:2'          OK
    ethernet/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:CONTROLLER'   OK
    ethernet/ipv6/icmpv6(type=135)-->'icmpv6_type=128,actions=output:2'          OK
    ethernet/vlan/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:2'      ERROR
        Received incorrect packet-in: ethernet(ethertype=34525)
    ethernet/vlan/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:CONTROLLER' ERROR
        Received incorrect packet-in: ethernet(ethertype=34525)
match: 30_ICMPV6_CODE
    ethernet/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:2'                OK
    ethernet/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:CONTROLLER'       OK
    ethernet/ipv6/icmpv6(code=1)-->'icmpv6_code=0,actions=output:2'                OK
    ethernet/vlan/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:2'          ERROR
        Received incorrect packet-in: ethernet(ethertype=34525)
    ethernet/vlan/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:CONTROLLER' ERROR
        Received incorrect packet-in: ethernet(ethertype=34525)

```

--- Test end ---

--- Test report ---

Received incorrect packet-in(4)

match: 29_ICMPV6_TYPE	ethernet/vlan/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:2'
match: 29_ICMPV6_TYPE	ethernet/vlan/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:CONTROLLER'
match: 30_ICMPV6_CODE	ethernet/vlan/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:2'
match: 30_ICMPV6_CODE	ethernet/vlan/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:CONTROLLER'

OK(6) / ERROR(4)

14.2 사용 방법

테스트 도구의 사용 방법을 설명합니다.

14.2.1 테스트 패턴

시험하려는 테스트 패턴에 따라 적절히 테스트 패턴 파일을 만들어야 있습니다.

테스트 패턴 파일 확장자를 「.json」고 말했다 텍스트 파일입니다. 다음의 형식으로 작성합니다.

```
[  
    "xxxxxxxxxxxx",                      # 시험 항목 이름  
    {  
        "description": "xxxxxxxxxxxx", # 시험 내용 설명  
        "prerequisite": [  
            {  
                "OFPFlowMod": {...} # 등록하는 유클 항목 또는 미터 항목  
            },                      # (Ryu의 OFPFlowMod 또는 OFPMeterMod을  
            {                        # json 형식으로 작성)  
                "OFPMeterMod": {...} # 기대할 처리 결과가  
            },                      # 패킷 전송 (actions = output)의 경우  
            {...}                   # 출력 포트 번호에 「2」를 지정하십시오  
        ],  
        "tests": [  
            {  
                # 인가 패킷  
                # 1 번만 적용할지 일정 시간 연속하여 인가 계속 여부에 따라  
                # (A) (B) 중 하나를 설명  
                # (A) 1번 인가  
                "ingress": [  
                    "ethernet(...)", # (Ryu 패킷 타이브러리 생성자의 형식으로 작성)  
                    "ipv4(...)",  
                    "tcp(...)"  
                ],  
                # (B) 일정 시간 연속 인가  
                "ingress": {  
                    "packets": {  
                        "data": [  
                            "ethernet(...)", # (A) 와 동일  
                            "ipv4(...)",  
                            "tcp(...)"  
                        ],  
                        "pktps": 1000,      # 초당 인가하는 패킷 수를 지정  
                        "duration_time": 30 # 연속 인가 시간을 초 단위로 지정  
                    }  
                },  
  
                # 기대할 처리 결과  
                # 처리 결과의 종별에 따라 (a) (b) (c) (d) 중 하나를 설명  
                # (a) 패킷 전송 (actions = output : X)의 확인 시험  
                "egress": [           # 기대할 전송 패킷  
                    "ethernet(...)",  
                    "ipv4(...)",  
                    "tcp(...)"  
                ]  
                # (b) 패킷 인 (actions = CONTROLLER)의 확인 시험  
                "PACKET_IN": [       # 기대할 Packet-In 메시지  
                    "ethernet(...)",  
                    "ipv4(...)",  
                    "tcp(...)"  
                ]  
                # (c) table-miss 확인 시험  
                "table-miss": [      # table-miss이 되는 것을 기대하는 유클 테이블 ID  
                    0  
                ]  
                # (d) 패킷 전송 (actions = output : X) 때 처리량의 확인 시험  
                "egress": [  
                    "throughput": [  
]
```

```

{
    "OFPMatch":{      # 처리량 측정에
        ...
    },                # 보조 SW에 등록된
    "kbytes":1000     # 흐름 항목 Match 조건
    # 예상 처리량을 Kbytes 단위로 지정
},
{...},
{...}
]
},
{...},
{...}
],
# 시험1
{...},
# 시험2
# 시험3
{...}
]
}
ryu/tests/switch/of13

```

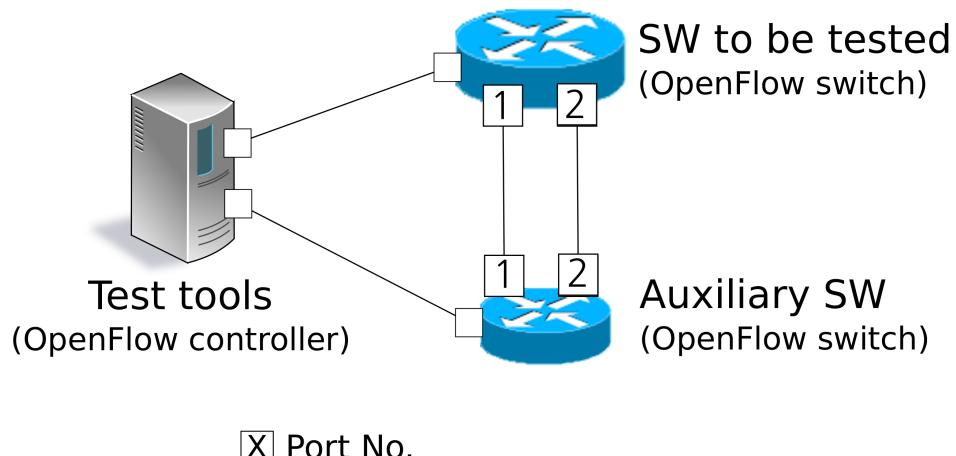
인가 패킷으로 「(B) 일정 시간 연속하여 인가」를, 기대하는 처리 결과로 「(d) 패킷 전송 (actions = output : X) 때 처리량의 확인 시험」을 각각 작성하여 시험 대상 SW의 처리량을 측정 할 수 있습니다.

주석: Ryu 소스 트리에는 샘플 테스트 패턴으로 OpenFlow1.3 FlowMod 메시지 match / actions 지정할 수 있는 매개 변수 및 MeterMod 메시지 매개 변수가 각각 정상적으로 작동하는지 확인 테스트 패턴 파일이 포함되어 있습니다.

ryu/tests/switch/of13

14.2.2 도구 실행 환경

테스트 도구 실행을 위한 환경은 다음과 같습니다.



Port No.

보조 스위치로 다음 동작을 완료 할 수 있다 OpenFlow 스위치가 필요합니다.

- actions=CONTROLLER의 흐름 항목 등록
- 처리량 측정을 위한 흐름 항목 등록
- actions=CONTROLLER의 흐름 항목에 의한 Packet-In 메시지 보내기
- Packet-Out 메시지 수신에 의한 패킷 전송

주석: Open vSwitch를 시험 대상 스위치 한 도구 실행 환경을 mininet에서 실현 환경 구축 스크립트가 Ryu 소스 트리에 포함되어 있습니다.

```
ryu/tests/switch/run_mininet.py
```

스크립트 예제를 「 테스트 도구 사용 예 」에 기재되어 있습니다.

14.2.3 테스트 도구 실행 방법

테스트 도구 Ryu 소스 트리에 제시되어 있습니다.

소스 코드	설명
ryu/tests/switch/tester.py	테스트 도구
ryu/tests/switch/of13	테스트 패턴 샘플
ryu/tests/switch/run_mininet.py	시험 환경 구축 스크립트

테스트 도구는 다음 명령을 실행합니다.

```
$ ryu-manager [--test-switch-target DPID] [--test-switch-tester DPID]
[--test-switch-dir DIRECTORY] ryu/tests/switch/tester.py
```

옵션	설명	기본값
-test-switch-target	시험되는 스위치의 데이터 경로 ID	00000000000000000001
-test-switch-tester	보조 스위치의 데이터 경로 ID	00000000000000000002
-test-switch-dir	테스트 패턴 파일의 디렉토리 경로	ryu/tests/switch/of13

주석: 테스트 도구 Ryu 응용 프로그램으로 ryu.base.app_manager.RyuApp을 상속 만들어 있기 때문에, 다른 Ryu 응용 프로그램과 마찬가지로 -verbose 옵션으로 디버깅 정보 출력 등에도 대응하고 있습니다.

테스트 도구를 시작한 후 시험 대상 스위치와 보조 스위치 컨트롤러 연결되면 지정된 테스트 패턴을 바탕으로 시험이 시작됩니다.

14.3 테스트 도구 사용 예

샘플 테스트 패턴과 원본 테스트 패턴을 이용한 테스트 도구의 실행 단계를 소개합니다.

14.3.1 샘플 테스트 패턴의 실행 단계

Ryu 소스 트리의 샘플 테스트 패턴 (ryu/tests/switch/of13)을 이용하여 FlowMod 메시지 match / actions의 대충의 동작 확인 및 MeterMod 메시지 동작 확인하는 절차를 보여줍니다.

이 단계에서는 시험 환경 시험 환경 구축 스크립트 (ryu/tests/switch/run_mininet.py)를 이용하여 구축하기로 합니다. 따라서 시험되는 스위치는 Open vSwitch입니다. VM 이미지 사용을 위한 환경 설정 및 로그인 방법 등은 「 스위칭 허브 」을 참조하십시오.

1. 시험 환경 구축

VM 환경에 로그인하고 시험 환경 구축 스크립트를 실행합니다.

```
ryu@ryu-vm:~$ sudo ryu/ryu/tests/switch/run_mininet.py
```

net 명령의 실행 결과는 다음과 같습니다.

```
mininet> net
c0
s1 lo: s1-eth1:s2-eth1 s1-eth2:s2-eth2
s2 lo: s2-eth1:s1-eth1 s2-eth2:s1-eth2
```

2. 테스트 도구 실행

테스트 도구 실행을 위한 컨트롤러의 xterm을 엽니다.

```
mininet> xterm c0
```

「Node: c0 (root)」의 xterm에서 테스트 도구를 실행합니다. 이때 테스트 패턴 파일 디렉토리로 샘플 테스트 패턴의 디렉토리 (`ryu/tests/switch/of13`)을 지정합니다. 또한 mininet 환경 시험 대상 스위치와 보조 스위치의 데이터 경로 ID는 각각 `-test-switch-target/-test-switch-tester` 옵션 기본값과 되어 있기 때문에 옵션을 생략합니다.

Node: c0:

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/of13 ryu/ryu/tests/swit
```

도구를 실행하면 다음과 같이 표시되고 시험되는 스위치와 보조 스위치가 컨트롤러에 연결될 때까지 기다립니다.

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/of13/ ryu/ryu/tests/swi
loading app ryu/ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu/ryu/tests/switch/tester.py of OfTester
target_dpid=0000000000000001
tester_dpid=0000000000000002
Test files directory = ryu/ryu/tests/switch/of13/
instantiating app ryu.controller.ofp_handler of OFPHandler
--- Test start ---
waiting for switches connection...
```

시험 대상 스위치와 보조 스위치가 컨트롤러에 연결되면 시험이 시작됩니다.

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/of13/ ryu/ryu/tests/swi
loading app ryu/ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu/ryu/tests/switch/tester.py of OfTester
target_dpid=0000000000000001
tester_dpid=0000000000000002
Test files directory = ryu/ryu/tests/switch/of13/
instantiating app ryu.controller.ofp_handler of OFPHandler
--- Test start ---
waiting for switches connection...
dpid=0000000000000002 : Join tester SW.
dpid=0000000000000001 : Join target SW.
action: 00_OUTPUT
    ethernet/ipv4/tcp-->'actions=output:2'          OK
    ethernet/ipv6/tcp-->'actions=output:2'          OK
    ethernet/arp-->'actions=output:2'                OK
action: 11_COPY_TTL_OUT
    ethernet/mpls(ttl=64)/ipv4(ttl=32)/tcp-->'eth_type=0x8847,actions=copy_ttl_out,output
        Failed to add flows: OFPErrorMsg[type=0x02, code=0x00]
    ethernet/mpls(ttl=64)/ipv6(hop_limit=32)/tcp-->'eth_type=0x8847,actions=copy_ttl_out,
        Failed to add flows: OFPErrorMsg[type=0x02, code=0x00]
...

```

`ryu/tests/switch/of13` 부하의 모든 샘플 테스트 패턴의 시험 가 완료되면 테스트 도구는 종료됩니다.

<참고>

샘플 테스트 패턴 파일 목록

`match/actions`의 각 설정 항목에 해당하는 흐름 항목을 등록하고 흐름 항목에 `match` (또는 `match하지 않는`) 여러 패턴의 패킷을 인가하는 테스트 패턴과 일정 빈도 이상 인가에 대해 삭제 또는 우선 순위 변경할 미터 항목을 등록하고 계량 항목에 `match` 패킷을 연속적으로 인가하는 테스트 패턴이 준비되어 있습니다.

```

ryu/tests/switch/of13/action:
00_OUTPUT.json          20_POP_MPLS.json
11_COPY_TTL_OUT.json    23_SET_NW_TTL_IPv4.json
12_COPY_TTL_IN.json     23_SET_NW_TTL_IPv6.json
15_SET_MPLS_TTL.json    24_DEC_NW_TTL_IPv4.json
16_DEC_MPLS_TTL.json    24_DEC_NW_TTL_IPv6.json
17_PUSH_VLAN.json       25_SET_FIELD
17_PUSH_VLAN_multiple.json 26_PUSH_PBB.json
18_POP_VLAN.json        26_PUSH_PBB_multiple.json
19_PUSH_MPLS.json       27_POP_PBB.json
19_PUSH_MPLS_multiple.json 27_POP_PBB.json

ryu/tests/switch/of13/action/25_SET_FIELD:
03_ETH_DST.json         14_TCP_DST_IPv4.json   24_ARP_SHA.json
04_ETH_SRC.json          14_TCP_DST_IPv6.json  25_ARP_THA.json
05_ETH_TYPE.json         15_UDP_SRC_IPv4.json  26_IPV6_SRC.json
06_VLAN_VID.json        15_UDP_SRC_IPv6.json  27_IPV6_DST.json
07_VLAN_PCP.json        16_UDP_DST_IPv4.json  28_IPV6_FLABEL.json
08_IP_DSCP_IPv4.json    16_UDP_DST_IPv6.json  29_ICMPV6_TYPE.json
08_IP_DSCP_IPv6.json    17_SCTP_SRC_IPv4.json 30_ICMPV6_CODE.json
09_IP_ECN_IPv4.json     17_SCTP_SRC_IPv6.json 31_IPV6_ND_TARGET.json
09_IP_ECN_IPv6.json     18_SCTP_DST_IPv4.json 32_IPV6_ND_SLL.json
10_IP_PROTO_IPv4.json   18_SCTP_DST_IPv6.json 33_IPV6_ND_TLL.json
10_IP_PROTO_IPv6.json   19_ICMPV4_TYPE.json   34_MPLS_LABEL.json
11_IPV4_SRC.json         20_ICMPV4_CODE.json   35_MPLS_TC.json
12_IPV4_DST.json         21_ARP_OP.json      36_MPLS_BOS.json
13_TCP_SRC_IPv4.json    22_ARP_SPA.json     37_PBB_ISID.json
13_TCP_SRC_IPv6.json    23_ARP_TPA.json     38_TUNNEL_ID.json

ryu/tests/switch/of13/match:
00_IN_PORT.json          13_TCP_SRC_IPv4.json  25_ARP_THA.json
02_METADATA.json          13_TCP_SRC_IPv6.json  25_ARP_THA_Mask.json
02_METADATA_Mask.json    14_TCP_DST_IPv4.json  26_IPV6_SRC.json
03_ETH_DST.json           14_TCP_DST_IPv6.json  26_IPV6_SRC_Mask.json
03_ETH_DST_Mask.json     15_UDP_SRC_IPv4.json  27_IPV6_DST.json
04_ETH_SRC.json           15_UDP_SRC_IPv6.json  27_IPV6_DST_Mask.json
04_ETH_SRC_Mask.json     16_UDP_DST_IPv4.json  28_IPV6_FLABEL.json
05_ETH_TYPE.json          16_UDP_DST_IPv6.json  29_ICMPV6_TYPE.json
06_VLAN_VID.json         17_SCTP_SRC_IPv4.json 30_ICMPV6_CODE.json
06_VLAN_VID_Mask.json   17_SCTP_SRC_IPv6.json 31_IPV6_ND_TARGET.json
07_VLAN_PCP.json         18_SCTP_DST_IPv4.json 32_IPV6_ND_SLL.json
08_IP_DSCP_IPv4.json    18_SCTP_DST_IPv6.json 33_IPV6_ND_TLL.json
08_IP_DSCP_IPv6.json    19_ICMPV4_TYPE.json   34_MPLS_LABEL.json
09_IP_ECN_IPv4.json     20_ICMPV4_CODE.json   35_MPLS_TC.json
09_IP_ECN_IPv6.json     21_ARP_OP.json      36_MPLS_BOS.json
10_IP_PROTO_IPv4.json   22_ARP_SPA.json     37_PBB_ISID.json
10_IP_PROTO_IPv6.json   22_ARP_SPA_Mask.json 37_PBB_ISID_Mask.json
11_IPV4_SRC.json         23_ARP_TPA.json     38_TUNNEL_ID.json
11_IPV4_SRC_Mask.json   23_ARP_TPA_Mask.json 38_TUNNEL_ID_Mask.json
12_IPV4_DST.json         24_ARP_SHA.json     39_IPV6_EXTHDR.json
12_IPV4_DST_Mask.json   24_ARP_SHA_Mask.json 39_IPV6_EXTHDR_Mask.json

ryu/tests/switch/of13/meter:
01_DROP_00_KBPS_00_1M.json 02_DSCP_REMARK_00_KBPS_00_1M.json
01_DROP_00_KBPS_01_10M.json 02_DSCP_REMARK_00_KBPS_01_10M.json
01_DROP_00_KBPS_02_100M.json 02_DSCP_REMARK_00_KBPS_02_100M.json
01_DROP_01_PKTPS_00_100.json 02_DSCP_REMARK_01_PKTPS_00_100.json
01_DROP_01_PKTPS_01_1000.json 02_DSCP_REMARK_01_PKTPS_01_1000.json
01_DROP_01_PKTPS_02_10000.json 02_DSCP_REMARK_01_PKTPS_02_10000.json

```

14.3.2 기존 테스트 패턴의 실행 단계

이제 원본의 테스트 패턴을 만들고 테스트 도구를 실행하는 방법을 설명합니다.

예를 들어, OpenFlow 스위치가 라우터 기능을 실현하기 위해 필요한 match / actions을 처리하는 기능을 가지고 있는지 확인하는 테스트 패턴을 만듭니다.

1. 테스트 패턴 생성

라우터가 라우팅 테이블에 따라 패킷을 전송하는 기능을 제공하는 다음 흐름 항목이 제대로 작동하는지 시험합니다.

match	actions
대상IP주소 범위 「192.168.30.0/24」	원본 MAC주소를 「aa:aa:aa:aa:aa:aa」로 수정 대상 MAC주소를 「bb:bb:bb:bb:bb:bb」로 수정 TTL 빼기 패킷 전송

이 테스트 패턴을 실행하는 테스트 패턴 파일을 만듭니다.

파일 이름: sample_test_pattern.json

```
[
    "sample: Router test",
    {
        "description": "static routing table",
        "prerequisite": [
            {
                "OFPFlowMod": {
                    "table_id": 0,
                    "match": {
                        "OFPMatch": {
                            "oxm_fields": [
                                {
                                    "OXMTlv": {
                                        "field": "eth_type",
                                        "value": 2048
                                    }
                                },
                                {
                                    "OXMTlv": {
                                        "field": "ipv4_dst",
                                        "mask": 4294967040,
                                        "value": "192.168.30.0"
                                    }
                                }
                            ]
                        }
                    }
                },
                "instructions": [
                    {
                        "OFPInstructionActions": {
                            "actions": [
                                {
                                    "OFPActionSetField": {
                                        "field": {
                                            "OXMTlv": {
                                                "field": "eth_src",
                                                "value": "aa:aa:aa:aa:aa:aa"
                                            }
                                        }
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        ]
    }
]
```

```

        "field": {
            "OXMTlv": {
                "field": "eth_dst",
                "value": "bb:bb:bb:bb:bb:bb"
            }
        }
    },
    {
        "OFPActionDecNwTtl": {}
    },
    {
        "OFPActionOutput": {
            "port": 2
        }
    }
],
"type": 4
}
]
}
],
"tests": [
{
    "ingress": [
        "ethernet(dst='22:22:22:22:22:22', src='11:11:11:11:11:11', ethertype=2048)",
        "ipv4(tos=32, proto=6, src='192.168.10.10', dst='192.168.30.10', ttl=64)",
        "tcp(dst_port=2222, option='\x00\x00\x00\x00', src_port=11111)",
        "'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'"
    ],
    "egress": [
        "ethernet(dst='bb:bb:bb:bb:bb:bb', src='aa:aa:aa:aa:aa:aa', ethertype=2048)",
        "ipv4(tos=32, proto=6, src='192.168.10.10', dst='192.168.30.10', ttl=63)",
        "tcp(dst_port=2222, option='\x00\x00\x00\x00', src_port=11111)",
        "'\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'"
    ]
}
]
}
]

```

2. 시험 환경 구축

시험 환경 구축 스크립트를 사용하여 시험 환경을 구축합니다. 절차는 샘플 테스트 패턴의 실행 단계 을 참조하십시오.

3. 테스트 도구 실행

컨트롤러 xterm에서 방금 만든 원래의 테스트 패턴 파일을 지정하여 테스트 도구를 실행합니다. 또한, -test-switch-dir 옵션은 디렉토리뿐만 아니라 파일을 직접 지정할 수 있습니다. 또한 송수신 패킷의 내용을 확인하기 위해 -verbose 옵션을 지정합니다.

Node: c0:

```
root@ryu-vm:~$ ryu-manager --verbose --test-switch-dir ./sample_test_pattern.json ryu/ryu/test
```

시험 대상 스위치와 보조 스위치가 컨트롤러에 연결되면 시험이 시작됩니다.

「dpid=0000000000000002 : receive_packet...」로깅에서 테스트 패턴 파일 egress 패킷으로 설정한 예상 출력 패킷 가 전송 된 것을 알 수 있습니다. 또한, 여기에서는 테스트 도구가 출력 한 로그만을 발췌하고 있습니다.

```

root@ryu-vm:~$ ryu-manager --verbose --test-switch-dir ./sample_test_pattern.json ryu/ryu/test
loading app ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/tests/switch/tester.py of OfTester
target_dpid=00000000000000000000000000000000
tester_dpid=00000000000000000000000000000000
Test files directory = ./sample_test_pattern.json

--- Test start ---
waiting for switches connection...

dpid=00000000000000000000000000000000 : Join tester SW.
dpid=00000000000000000000000000000000 : Join target SW.

sample: Router test

send_packet:[ethernet(dst='22:22:22:22:22:22', ethertype=2048, src='11:11:11:11:11:11'), ipv4(csum=54545454, dst='192.168.1.1', egress=[ethernet(dst='bb:bb:bb:bb:bb:bb', ethertype=2048, src='aa:aa:aa:aa:aa:aa')], ipproto=1, payload=[tcp(sport=1234, dport=80, data='GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n')]), priority=0, nw_dst=192.168.1.1]
egress:[ethernet(dst='bb:bb:bb:bb:bb:bb', ethertype=2048, src='aa:aa:aa:aa:aa:aa')], packet_in: []
dpid=00000000000000000000000000000000 : receive_packet[ethernet(dst='bb:bb:bb:bb:bb:bb', ethertype=2048, src='aa:aa:aa:aa:aa:aa')], static routing table OK
--- Test end ---

```

실제로 OpenFlow 스위치에 등록 된 흐름 항목은 다음과 같습니다. 테스트 도구에 의해 인가된 패킷 흐름 항목에 match하고 n_packets가 올라 가게되는 것을 알 수 있습니다.

Node: s1:

```

root@ryu-vm:~# ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=56.217s, table=0, n_packets=1, n_bytes=73, priority=0, ip,nw_dst=192.168.1.1
    actions=OUTPUT:1

```

14.3.3 오류 메시지 목록

이 도구에서 출력되는 오류 메시지 목록을 보여줍니다.

오류 메시지	설명
Failed to initialize flow tables: barrier request timeout.	지난번 시험 대상 SW의 흐름 항목 삭제에 실패 (Barrier Request Timeout)
Failed to initialize flow tables: [err_msg]	마지막 시험 대상 SW의 흐름 항목 삭제에 실패 (Failure Message)
Failed to initialize flow tables of tester_sw: barrier request timeout.	지난번 시험의 보조 SW의 흐름 항목 삭제에 실패 (Barrier Request Timeout)
Failed to initialize flow tables of tester_sw: [err_msg]	마지막 시험 보조 SW의 흐름 항목 삭제에 실패 (FlowMod Failure Message)
Failed to add flows: barrier request timeout.	시험 대상 SW에 대한 흐름 항목 등록에 실패 (Barrier Request Timeout)
Failed to add flows: [err_msg]	시험 대상 SW에 대한 흐름 항목 등록에 실패 (FlowMod Failure Message)
Failed to add flows to tester_sw: barrier request timeout.	보조 SW에 대한 흐름 항목 등록에 실패 (Barrier Request Timeout)
Failed to add flows to tester_sw: [err_msg]	보조 SW에 대한 흐름 항목 등록에 실패 (FlowMod Failure Message)
Failed to add meters: barrier request timeout.	시험 대상 SW에 대한 미터 항목 등록에 실패 (Barrier Request Timeout)
Failed to add meters: [err_msg]	시험 대상 SW에 대한 미터 항목 등록에 실패 (MeterMod Failure Message)
Added incorrect flows: [flows]	시험 대상 SW에 대한 흐름 항목 등록 확인 오류 (예기치 않은 흐름 항목)
Failed to add flows: flow stats request timeout.	시험 대상 SW에 대한 흐름 항목 등록 확인에 실패 (FlowStats Request Timeout)
Failed to add flows: [err_msg]	시험 대상 SW에 대한 흐름 항목 등록 확인에 실패 (FlowMod Failure Message)
Added incorrect meters: [meters]	시험 대상 SW에 대한 미터 항목 등록 확인 오류 (예기치 않은 미터 항목)
Failed to add meters: meter config stats request timeout.	시험 대상 SW에 대한 미터 항목 등록 확인에 실패 (MeterStats Request Timeout)
Failed to add meters: [err_msg]	시험 대상 SW에 대한 미터 항목 등록 확인에 실패 (MeterMod Failure Message)
Failed to request port stats from target: request timeout.	시험 대상 SW의 PortStats 가져 오지 (PortStats Request Timeout)
Failed to request port stats from target: [err_msg]	시험 대상 SW의 PortStats 가져 오지 (PortStats Request Failure Message)
Failed to request port stats from tester: request timeout.	보조 SW의 PortStats 가져 오지 (PortStats Request 시간 지연)
Failed to request port stats from tester: [err_msg]	보조 SW의 PortStats 가져 오지 (PortStats Request에 대한 Failure Message)

Table 14.1 – 이전 페이지에서 계속

오류 메시지	설명
Received incorrect [packet]	기대 한 출력 패킷의 수신 오류 (잘못된 패킷을 수신)
Receiving timeout: [detail]	기대 한 출력 패킷 수신에 실패 (시간 초과)
Faild to send packet: barrier request timeout.	패킷인가 실패 (Barrier Request 시간 제한)
Faild to send packet: [err_msg]	패킷인가 실패 (Packet-Out 대한 Error 메시지 수신)
Table-miss error: increment in matched_count.	table-miss 확인 오류 (흐름에 match하고 있다)
Table-miss error: no change in lookup_count.	table-miss 확인 오류 (패킷이 대상의 흐름 테이블에서 찾았지만 lookup_count가 바뀌지 않았다)
Failed to request table stats: request timeout.	table-miss 확인에 실패 (TableStats Request 시간 제한)
Failed to request table stats: [err_msg]	table-miss 확인에 실패 (TableStats Request에 대한 Error 메시지 수신)
Added incorrect flows to tester_sw: [flows]	보조 SW에 대한 흐름 항목 등록 확인 오류 (예기치 않은 흐름 항목 등록)
Failed to add flows to tester_sw: flow stats request timeout.	보조 SW에 대한 흐름 항목 등록 확인에 실패 (FlowStats Request 시간 제한)
Failed to add flows to tester_sw: [err_msg]	보조 SW에 대한 흐름 항목 등록 확인에 실패 (FlowStats Request에 대한 Error 메시지 수신)
Failed to request flow stats: request timeout.	처리량 확인 시 보조 SW에 대한 흐름 항목 등록 확인에 실패 (FlowStats Request 시간 제한)
Failed to request flow stats: [err_msg]	처리량 확인 시 보조 SW에 대한 흐름 항목 등록 확인에 실패 (FlowStats Request에 대한 Error 메시지 수신)
Received unexpected throughput: [detail]	상정 처리량에서 동떨어진 처리량을 측정
Disconnected from switch	시험 대상 SW 또는 보조 SW에서 링크 단선 발생

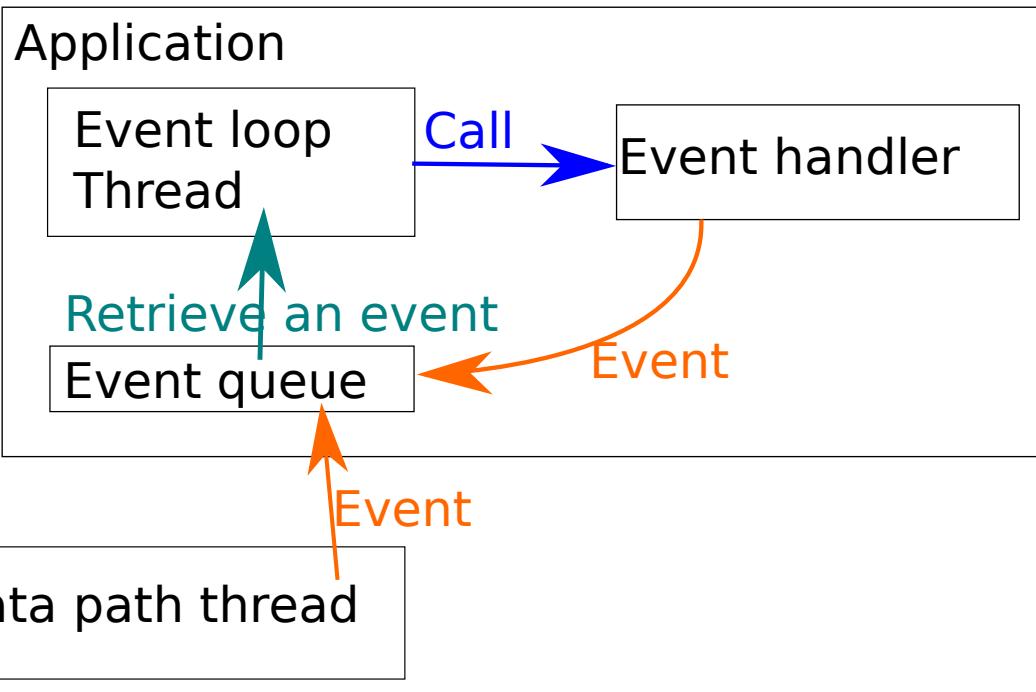
아키텍처

Ryu 아키텍처를 소개 합니다. 각 클래스의 사용법 등은 [API 레퍼런스](#)를 참조하십시오.

15.1 응용 프로그래밍 모델

Ryu 응용 프로그램 프로그래밍 모델을 설명합니다.

ryu-manager process



15.1.1 응용 프로그램

응용 프로그램은 `ryu.base.app_manager.RyuApp` 을 계승한 클래스입니다. 사용자 논리는 응용 프로그램이라고 합니다.

15.1.2 이벤트

이벤트는 `ryu.controller.event.EventBase` 를 상속한 클래스의 개체입니다. 응용 프로그램 간의 통신은 이벤트를 송수신 함으로써 가능합니다.

15.1.3 이벤트 큐

각 응용 프로그램은 이벤트 수신을 위한 큐를 하나 가지고 있습니다.

15.1.4 스레드

Ryu 는 `eventlet`을 사용한 다중 스레드로 동작 합니다. 스레드는 비선점형 이므로, 시간이 걸리는 처리를 행하는 경우에는 주의가 필요합니다.

이벤트 루프

응용 프로그램 당 한 개의 스레드가 자동으로 생성 됩니다. 이 스레드는 이벤트 루프를 실행 합니다. 이벤트 루프는 이벤트 큐에 이벤트가 있으면 꺼내 해당 이벤트 처리기 (아래) 를 호출합니다.

추가 스레드

`hub.spawn` 함수를 사용하여 추가 스레드를 만들고 응용 프로그램 별 처리를 할 수 있습니다.

eventlet

`eventlet` 기능을 애플리케이션에서 직접 사용될 수 있지만, 비추천입니다. 가능하다면 `hub` 모듈 제공하는 래퍼를 사용하도록 하십시오.

15.1.5 이벤트 처리기

응용 프로그램 클래스의 메서드를 `ryu.controller.handler.set_ev_cls` 장식으로 한정하여 이벤트 처리기를 정의 할 수 있습니다. 이벤트 처리기는 지정된 형식의 이벤트가 발생했을 때 응용 프로그램 이벤트 루프에서 호출됩니다.

컨트리뷰션

오픈 소스 소프트웨어의 묘미 중 하나는 자체 개발에 참여할 수 있습니다. 이 장에서는 Ryu의 개발에 참여하는 방법을 소개 합니다.

16.1 개발 체제

Ryu의 개발은 메일링리스트를 중심으로 진행되고 있습니다. 우선은 메일링리스트에 가입하는 것부터 시작 합시다.

<https://lists.sourceforge.net/lists/listinfo/ryu-devel>

메일링리스트의 교환은 기본적으로 영어로 진행 됩니다. 사용법 등 의문이 있거나 결합 것으로 보인다 같은 행동에 조우했을 때, 이메일을 보내는 것을 망설일 필요는 없습니다. 오픈 소스 소프트웨어를 사용하는 것 자체가 프로젝트에 중요한 기여하기 때문입니다.

16.2 개발 환경

이 섹션에서는 Ryu의 개발에 필요한 환경 및 고려 사항에 대해 설명 합니다.

16.2.1 Python

Ryu는 Python 2.6 이상을 지원합니다. 즉, Python 2.7에서만 사용 가능한 구문 등을 사용하지 마십시오.

Python 3.0 이상은 당분간 지원되지 않습니다. 하지만 소스 코드는 향후 변경이 가능한 적게 않아도되는 기술을 유의하면 좋을 것입니다.

16.2.2 코딩 스타일

Ryu 소스 코드 PEP8하는 코딩 스타일을 준수하고 있습니다. 후술하는 패치 발송 시에는 그 내용이 PEP8을 준수하고 있는지 미리 확인하십시오.

<http://www.python.org/dev/peps/pep-0008/>

또한, 소스 코드가 PEP8을 준수하는지 확인하려면 테스트 섹션 소개하는 스크립트와 함께 검사기를 이용할 수 있습니다.

<https://pypi.python.org/pypi/pep8>

16.2.3 테스트

Ryu에는 몇 가지 자동화 된 테스트가 있지만 가장 간단하고 많이 사용되는 것은 Ryu만으로 완결하는 단위 테스트입니다. 후술하는 패치 발송 시에는 변경 사항 때문에 단위 테스트 실행 실패하지 않는 것을 미리 확인 하십시오. 또한 새로 추가된 소스 코드는 가능한 한 단위 테스트를 설명하는 것이 바람직 할 것입니다.

```
$ cd ryu/
$ ./run_tests.sh
```

16.3 패치 쓰기

기능 추가 및 버그 수정 등으로 저장소의 소스 코드를 변경하는 때 변경 사항을 패치 한 후, 우편으로 보냅니다. 큰 변경은 미리 메일링리스트에서 논의되고 있다고 바람직한 것입니다.

주석: Ryu 소스 코드 저장소는 GitHub에 존재하지만, 풀 요청을 이용한 개발 프로세스가 아님에 주의 하십시오.

송부하는 패치 형식은 Linux 커널 개발에서 사용되는 스타일이 상정되고 있습니다. 이 섹션에서는 이 스타일 패치를 메일링리스트에 쓰기 까지의 일례를 소개하고 있습니다만, 더 자세한 내용은 관련 문서를 참조 하십시오.

<http://lxr.linux.no/linux/Documentation/SubmittingPatches>

그럼 단계를 소개 합니다.

1. 소스 코드를 체크 아웃

우선 Ryu의 소스 코드를 체크 아웃 합니다. GitHub에서 소스 코드를 fork하여 자신의 작업 저장소를 만들어서 상관 없지만, 단순화하기 위해 원본을 그대로 사용한 예제입니다.

```
$ git clone https://github.com/osrg/ryu.git $ cd ryu/
```

2. 소스 코드를 변경

Ryu 소스 코드에 필요한 사항을 변경합니다. 작업 구분할 때는 변경 내용을 커밋 합니다.

```
$ git commit -a
```

3. 패치를 만들기

변경 내용의 차등을 패치 합니다. 패치는 Signed-off-by: 행을 붙이는 것을 잊지 마십시오. 이 서명은 당신이 제출한 패치가 오픈 소스 소프트웨어 라이선스, 문제없는 것을 선언 합니다.

```
$ git format-patch origin -s
```

4. 패치 쓰기

완성된 패치의 내용이 올바른지 확인한 후, 우편으로 보냅니다. 귀하의 우편물에서 직접 보낼 수도 있지만 git-send-email(1)을 사용하는 것으로 대화식으로 처리할 수 있습니다.

```
$ git send-email 0001-sample.patch
```

5. 응답 대기

패치에 대한 응답을 기다립니다. 그대로 받아 들여지는 경우도 있습니다만, 지적 사항 등이 있으면 내용을 수정하여 다시 보낼 필요가 있을 것입니다.

도입 사례

이 장에서는 Ryu을 이용한 서비스 / 제품의 사례를 소개합니다.

17.1 Stratosphere SDN Platform (스트라토스 피어)

Stratosphere SDN Platform (이하 SSP)는 스트라토스 피어 사의 개발 소프트웨어 제품입니다. SSP를 이용하여 VXLAN, STT, MPLS 같은 터널링 프로토콜을 사용하여 가장자리 오버레이 형식의 가상 네트워크를 만들 수 있습니다.

각 터널링 프로토콜은 VLAN과 상호 변환됩니다. 각 터널링 프로토콜 식별자가 VLAN 12 비트보다 크기 때문에 VLAN을 직접 사용하는 것보다 많은 L2 세그먼트를 관리 할 수 있습니다. 또한 SSP는 OpenStack과 CloudStack 같은 IaaS 소프트웨어와 함께 사용할 수 있습니다.

SSP는 기능을 수행하는 OpenFlow를 사용하고 있으며, 버전 1.1.4에서는 컨트롤러에 Ryu을 채용하고 있습니다. 이유는 먼저 OpenFlow1.1 이상으로 대응을 들 수 있습니다. SSP를 MPLS에 대응 시키는데 프로토콜 수준에서의 지원이 OpenFlow1.1 이후 지원하는 프레임 워크의 도입이 생각되었습니다.

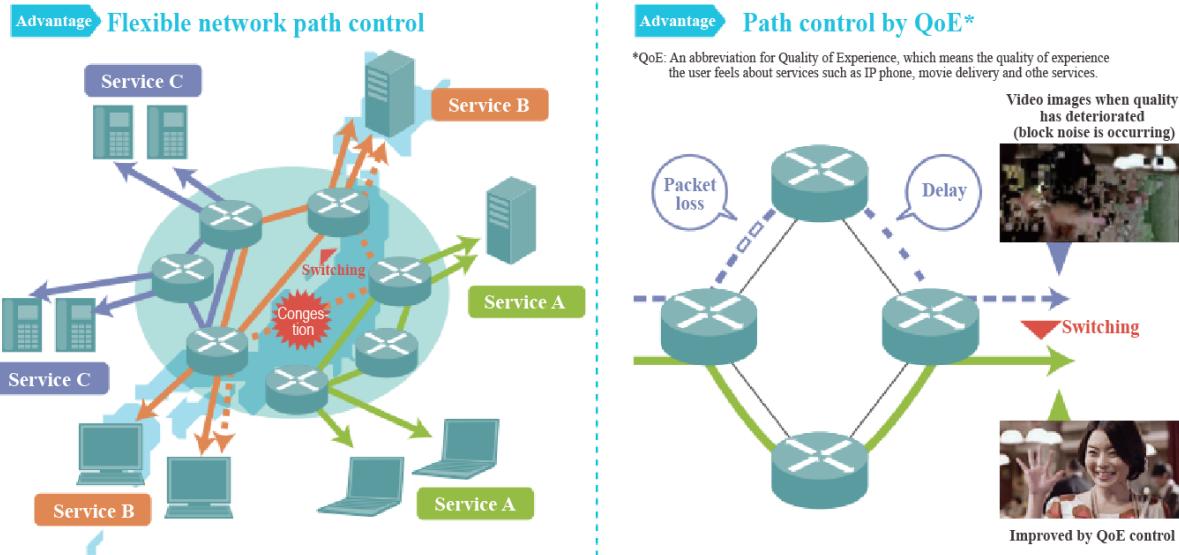
주석: OpenFlow 프로토콜 자체의 지원과는 별도로 구현이 선택적인 항목은 사용하는 OpenFlow 스위치 측의 지원 상황도 충분히 고려해야합니다.

또한 개발 언어로 Python을 사용할 수 있는 점도 들 수 있습니다. 스트라토스 피어의 개발 Python을 적극적으로 사용하고 있으며, SSP 많은 부분이 Python으로 작성되어 있습니다. Python 자체 기술력의 높이와 친숙한 언어를 유효하다 개발 효율의 향상을 기대할 수 있었습니다.

소프트웨어는 여러 Ryu 응용 프로그램 만들고 REST API를 통해 SSP의 다른 구성 요소와 상호 작용합니다. 소프트웨어를 기능 단위로 여러 응용 프로그램에 분할 할 수는 전망 좋은 소스 코드를 유지하는데 필수적이었습니다.

17.2 SmartSDN Controller (NTT 컴웨어)

「SmartSDN Controller」는 기존의 자율 분산 제어에 변하는 네트워크 집중 제어 기능 (네트워크 가상화 / 최적화 등)을 제공하는 SDN 컨트롤러입니다.



「SmartSDN Controller」은 다음 두 가지의 특징을 가지고 있습니다.

1. 가상 네트워크를 통한 유연한 네트워크 라우팅

동일한 실제 네트워크에 여러 개의 가상 네트워크를 구축하여, 사용자의 요구에 유연한 네트워크 환경을 제공하고 시설 활용에 따라 서비스 비용의 절감을 가능하게 합니다. 또, 지금까지 개별적으로 정보 설정하고 스위치 라우터를 중앙에서 관리함으로써 네트워크 전체를 파악하고 고장이나 네트워크의 트래픽 상황에 맞는 유연한 경로 변경을 가능하게 합니다.

서비스 이용자의 체감 품질(「QoE」:Quality of Experience)에 주목하고, 통신이 흐르고 있는 네트워크의 품질(대역폭, 지연, 손실, 움직임 등)에서 체감 품질(QoE)을 확인하고 더 나은 경로로 우회하여 서비스 품질 안정 유지를 실현합니다.

2. 고급 보수 운용 기능으로 네트워크의 신뢰성 확보

컨트롤러의 고장 발생시에도 서비스를 계속하기 위해 중복 구성을 실현하고 있습니다. 또한 거점 사이를 흐르는 통신 패킷을 유사적으로 만들고 경로에 흘리는 것으로 OpenFlow 사양에 명시된 표준 모니터링 기능으로는 인식 할 수 없는 경로상의 고장의 조기 발견 및 각종 시험(소통 확인, 경로 확인 등)을 가능하게 합니다.

또한 네트워크 설계, 네트워크 상태 확인은 GUI를 통해 시각화하고 보수의 스킬 레벨에 의하지 않는 운용을 가능하게 하고, 네트워크 운영 비용을 절감합니다.

「SmartSDN Controller」의 개발에 있어서는 다음의 조건을 만족 OpenFlow의 프레임워크를 선정 할 필요가 있었습니다.

- OpenFlow 사양을 포괄적으로 지원할 수 있는 프레임워크입니다
- OpenFlow 버전 업에 추종을 계획하고 있기 때문에, 비교적 빨리 추종 대응이 되는 프레임워크입니다

그 중에 Ryu는

- OpenFlow의 각 버전의 기능을 두루 지원
- OpenFlow 버전 업에 추종 대응이 빠르다. 또한 개발 커뮤니티 활동적이고 버그에 대한 대응이 빠른
- 샘플 코드 / 문서가 충실히다

등의 특징을 가지고 있기 때문에 프레임워크로 적합하다고 판단하여 채택했습니다.